

AIF2 Low-level Driver

User's Guide

Applies to Product Release: 01.02.00.00:
Publication Date: June, 2015

Document License

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/3.0/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Contributors to this document

Copyright (C) 2011 Texas Instruments Incorporated - <http://www.ti.com/>

Texas Instruments, Incorporated
5 Chemin des Presses, 4 allée Technopolis
06800 Cagnes sur Mer,
FRANCE



Table of Contents

| | | |
|-------|---|----|
| 1. | About AIF2 LLD..... | 5 |
| 1.1 | Benefits | 5 |
| 1.2 | Features | 6 |
| 2. | Software overview | 6 |
| 2.1 | Architecture | 7 |
| 2.2 | Using AIF2 LLD for Wcdma traffic | 7 |
| 2.3 | Using AIF2 LLD for LTE traffic | 8 |
| 2.4 | Using AIF2 LLD for dual mode traffic | 8 |
| 2.5 | Using AIF2 LLD to transfer generic packets | 8 |
| 2.6 | Using AIF2 LLD for CPRI fast c&m traffic..... | 9 |
| 3. | Programming guidelines | 9 |
| 3.1 | Driver configuration..... | 9 |
| 3.1.1 | AIF_ConfigObj data structure | 9 |
| 3.2 | CPRI and OBSAI link configuration | 10 |
| 3.2.1 | Antenna container configuration..... | 11 |
| 3.2.2 | CPRI flexible antenna container packing (LTE only)..... | 12 |
| 3.3 | AIF2 physical Timings parameters | 13 |
| 3.3.1 | Pi value..... | 13 |
| 3.3.2 | Using link retransmission (RT)..... | 14 |
| 3.4 | DIO engines..... | 16 |
| 3.4.1 | Ingress and Egress parameters | 16 |
| 3.4.2 | DIO and RAC operation | 17 |
| 3.4.3 | DIO and TAC operation | 18 |
| 3.4.4 | DIO parameters..... | 19 |
| 3.4.5 | LLD restriction | 19 |
| 3.4.6 | DIO duplicate ingress traffic on DIO 2..... | 19 |
| 3.5 | PKTDMA channels | 20 |
| 3.5.1 | Memory region | 21 |
| 3.5.2 | Antenna containers..... | 21 |

| | | |
|-------|--|----|
| 3.5.3 | Pushing and recycling packets | 23 |
| 3.5.4 | CPRI Fast C&M control words | 23 |
| 3.6 | Physical and Radio timers | 25 |
| 3.6.1 | Using pre-defined AT events and counters..... | 25 |
| 3.6.2 | Adding application-specific AT events | 26 |
| 3.6.3 | Triggering physical and radio timers..... | 27 |
| 3.7 | AIF2 exceptions | 27 |
| 4. | AIF2 hardware initialization | 28 |
| 4.1 | Sequence of AIF2 LLD calls at start-up..... | 28 |
| 4.2 | AT event programming at run-time | 30 |
| 4.3 | Re-initialization | 30 |
| 5. | AIF2 LLD testing..... | 30 |
| 6. | AIF2 LLD example configuration | 30 |
| 6.1 | testAIF2_Wcdma4x..... | 30 |
| 6.2 | testAIF2_cpriWcdmaCheckRF | 34 |
| 6.3 | testAIF2_cpriRAC | 36 |
| 6.4 | testAIF2_cpriDLte..... | 36 |
| 7. | Different AxC sampling rate on a same CPRI link..... | 37 |
| 7.1 | Dual LTE..... | 37 |
| 7.2 | Dual mode..... | 39 |
| 8. | Software implementation of CPRI FastCM over AIF2 | 43 |
| 8.1 | Introduction | 43 |
| 8.2 | Software workaround implementation | 44 |
| 8.2.1 | AIF2 configuration in Null delimiter mode..... | 45 |
| 8.2.2 | TX 4B 5B encoder | 45 |
| 8.2.3 | RX 5B 4B Decoder..... | 47 |

List of figures

| | |
|---|----|
| Figure 1: Software components..... | 6 |
| Figure 2: Software architecture | 7 |
| Figure 3: Pi and Delta Timing example | 15 |
| Figure 4: Link forwarding and retransmission between 2 devices | 15 |
| Figure 5: AIF2 to RAC DIO configuration..... | 17 |
| Figure 6: TAC to AIF2 DIO configuration | 18 |
| Figure 7: Packet DMA and AIF2..... | 20 |
| Figure 8: Packet DMA queues (Q) and free descriptor queues (FDQ) | 21 |
| Figure 9: AIF2 driver initialization sequence..... | 29 |
| Figure 10: Wcdma egress DIO data in SoC memory | 32 |
| Figure 11: Wcdma ingress DIO data in SoC memory | 33 |
| Figure 12: Wcdma egress DIO data in SoC memory | 35 |
| Figure 13: Wcdma ingress DIO data in SoC memory | 36 |
| Figure 14: dual LTE workaround for LTE 5 MHz..... | 38 |
| Figure 15: LLD implementation of dual mode on a single CPRI link | 40 |
| Figure 16: CPRI basic frame format for dual mode on a single CPRI link..... | 41 |
| Figure 17: Ethernet frame structure | 43 |
| Figure 18: IEEE 802.3 data transmission format..... | 43 |
| Figure 19: Ethernet frame preamble | 44 |

User's guide

AIF2 LLD version 01.02.00.00

This document describes how to use the Antenna Interface Low Level Driver.

1. About AIF2 LLD

The AIF2 low-level driver is meant to be used by drivers and application that interfaces with AIF2, QMSS and CPPI IPs. This AIF2 low-level driver aims at generalizing the configuration of AIF2 for different modes (CPRI/OBSAI/Generic packet, WCDMA/LTE/Dual mode). It should be noted that AIF2 LLD support a single instance of the driver run from only one of the KeyStone-I or KeyStone-II SOC DSP cores. An AIF2 LLD test example showing how to use a single LLD instance across multiple cores is provided in the AIF2 LLD package.

The first goal of the AIF2 LLD is to provide customers with a “functional layer” or an abstraction of the AIF2 configuration complexity. That means that within a short amount of configuration parameters / API calls, the end user can configure AIF2 for a specific high level scenario. Enough flexibility has been put in the LLD at this point to achieve that goal. The current LLD implementation actually doesn't prevent the end user from overriding the “pre-defined” parameters. APIs have been split in such a way that first the Functional layer high level structure (Aif2FI_Setup) gets populated (AIF_initHw):

```
// initialization function for the AIF2 H/W FL structure (can still be overridden afterwards)
```

```
AIF_initHw(&aifObj);
```

And then the FL configuration to AIF2 HW registers in a separate API call. That means aifObj.hAif2Setup can still be altered in between with the “end user” very specific needs:

```
AIF_startHw(&aifObj);
```

1.1 Benefits

The idea of providing a low level driver is to abstract the programming complexity of AIF2 IP while still allowing the drivers and applications to control every aspect of AIF2.

Debug options and tools within AIF2 LLD and its utilities are provided to ease debug this IP on target and help the AIF2 end user application. To name some of these, AIF2 LLD includes:

- AIF2 HW exceptions: LLD has a set of APIs to enable/gather/print exceptions
- AIF2 DIO replicate on dio_2: LLD can duplicate DIO traffic on 2 different engines. This helps when customer want to debug antenna data carried to the RAC accelerators

- AIF2 FL dumper: it is created from a perl script that generates a C file given a pointer to a FL structure (note this is not specific to AIF2). So within AIF2 LLD utilities, the end user can dump from the high level FL structure (Aif2FI_Setup) all structure fields. This gives an output file that turns out to be very useful to compare AIF2 configurations between a working and non-working use case.

1.2 Features

AIF2 LLD feature set is evolving over time. WCDMA, LTE FDD and Generic packet, mainly in CPRI mode, have been the primary features of initial releases. Since then, the AIF2 LLD has added support for LTE/WCDMA simultaneous dual mode, LTE TDD, and OBSAI protocol (WCDMA/LTE).

2. Software overview

The AIF2 LLD main data structures contain the properties of the AIF2 hardware entities that are of interest for an application that aims at programming AIF2 with high-level parameters, or parameters that are significant enough for a user application. For a wireless base station application that makes use of the KeyStone-I / KeyStone-II wireless-dedicated accelerators and implements high channel density and data throughput, the AIF2 LLD implements enough flexibility and scales over all AIF2 hardware resources in the following way:

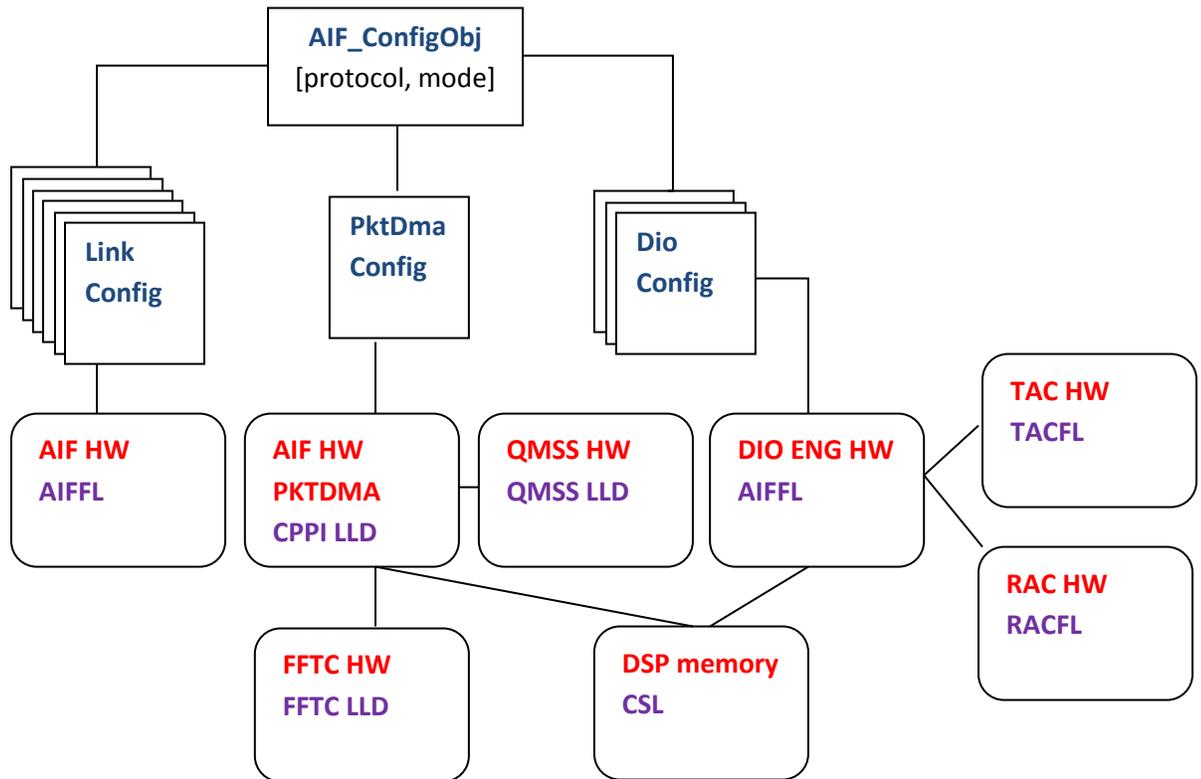


Figure 1: Software components

2.1 Architecture

This section explains the overall architecture of AIF2 LLD (starting from v1.2 release, for ARM and DSP):

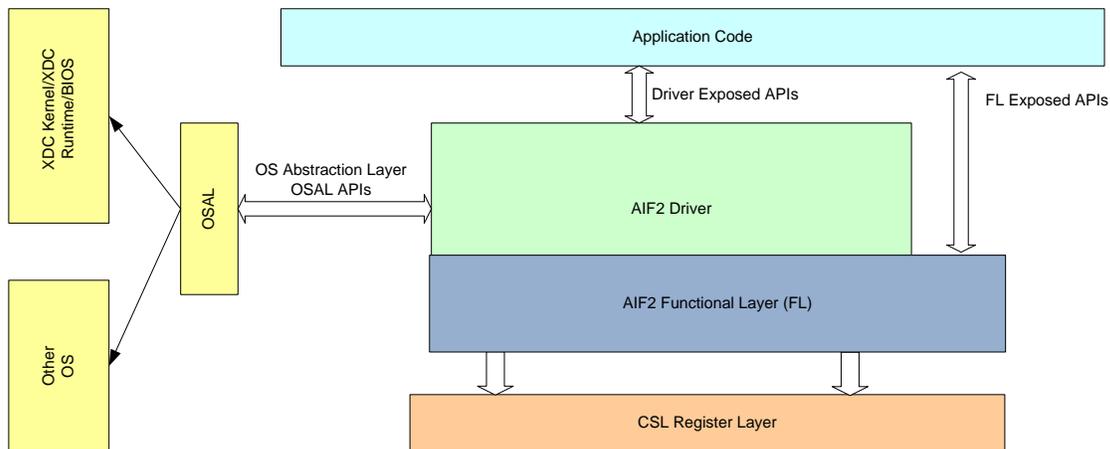


Figure 2: Software architecture

The AIF2 LLD architecture diagram illustrates the following key components:

1.) AIF2 Driver

This is the core AIF2 driver. The driver exposes a set of APIs which are used by the application layer to configure, monitor, reconfigure and reset the AIF2 peripheral module. The driver also exposes a set of OS abstraction APIs which are used to ensure that the driver is OS independent and portable. The AIF2 driver uses its AIF2 functional layer for all AIF2 MMR access.

2.) Application Code

This is the user of the driver and its interface with the driver is through the APIs set. Application users use the driver APIs to configure, monitor, reconfigure and reset the AIF2 module.

3.) Operating System Abstraction Layer (OSAL)

The AIF2 LLD is OS independent and exposes all the operating system callouts via this OSAL layer.

4.) AIF2 Functional Layer

The AIF2 Functional Layer is used to setup AIF2 HW via a top-level FL configuration structure which exposes all the AIF2 low-level fields. It also offers a set of specific commands and queries for each AIF2 sub-modules.

5.) CSL Register Layer

The CSL register layer is the IP block memory mapped registers which are generated by the IP owner. CSLR names are matching the IP HW specification. The AIF2 FL accesses the MMR registers via CSLR.

2.2 Using AIF2 LLD for Wcdma traffic

The AIF2 LLD supports setting a WCDMA communication between DSP and radio equipment via multiple high-speed SerDes links. They are two industry communication protocols that can be used in this application: OBSAI and CPRI. The AIF2 LLD supports both Uplink and Downlink for direct connection to RAC and TAC, and can be configured to use one or multiple links. AIF2 LLD initially supports the following link rates: {4x,8x}.

AIF2 was designed to use up to 3 DIO engines for WCDMA traffic. For each enabled link, the end user associates one of the DIO engines. The DIO engines essentially work with circular buffers of antenna samples, and act as masters on the SoC interconnect, allowing seamless connections with Wcdma TAC and RAC chip-rate accelerators, and also the DSP memory subsystem, including local, shared, and DDR memories. AIF2 LLD allows end user to specify the placement and depth of the circular buffers associated to each of the DIO engines.

AIF2 LLD also supports daisy chain topologies, by implementing proper timing on these links that make use of the AIF2 re-transmission feature.

2.3 Using AIF2 LLD for LTE traffic

The AIF2 LLD works together with CPPI and QMSS LLDs to support the transport of LTE antenna samples between the radio equipment and the KeyStone FFTC accelerators. AIF2 HW was designed to work well with KeyStone PKTDMA and Queue Manager for that purpose. The AIF2 LLD can configure 1.4, 3, 5, 10, 20 and 40 MHz LTE traffic.

The AIF2 LLD offers a flexible configuration of hardware queues and packet dma channels/receive flows for each enabled antenna carrier across all links. This allows the end user to have full control about the application architecture around outgoing/incoming LTE symbol packets.

For LTE TDD, AIF2 LLD defines the different UL-DL configurations as well as the different special subframe configurations.

2.4 Using AIF2 LLD for dual mode traffic

As stated above here, AIF2 is designed to support WCDMA antenna traffic by using DIO engines, which can interface directly with RAC and TAC, and to support LTE antenna traffic by using PKTDMA, which can interface with FFTC directly. An important point is that AIF2 DIO mode and PKTDMA mode cannot co-exist. The solution chosen to support dual mode (simultaneous antenna traffic of WCDMA and LTE) is to use DIO to support both LTE and WCDMA.

Using the DIO approach for both modes, it is still possible to have zero CPU intervention to connect AIF2 and FFTC for the LTE antenna traffic. On ingress side, since antenna data placement is regular in DIO circular buffers, packet descriptors can be pre-configured to point to the corresponding position for each symbol. EDMA is then used to pop the pre-arranged descriptor from the queue and push it to the input queue of FFTC to achieve zero CPU intervention. On egress side, Rx/output free descriptor queue of FFTC can be arranged to have the descriptors pointing to the buffers that matches the DIO data placement.

2.5 Using AIF2 LLD to transfer generic packets

The AIF2 has a special mode called generic packet for inter-device communication. It allows using AIF2 to transfer generic digital packets between DSPs or between a DSP and a FPGA. They are two standardized communication protocols that can be used in this mode: OBSAI and CPRI. The AIF2 LLD supports only 16-bit data width.

Packet mode does not require time synchronization. As opposed to IQ data antenna traffic, Packet mode allows the user to send data packets asynchronously. Generic data can be transferred over control message slot (for OBSAI) or control words (for CPRI) in parallel with antenna stream. If one link is not used for antenna stream, generic data can also be transferred over AxC (Antenna stream Carrier) slots. Generic packet mechanism is commonly used to group multiple words or messages together to form larger packets, or to utilize unused bandwidth of the link.

OBSAI support generic message in nature. To support generic data transfer, CPRI need additional delimiter, 4B/5B or NULL delimiter can be used for generic data over control words, since 4B/5B introduce more overhead, so NULL delimiter can be used for generic data over control words. But for generic data over AxC slots in CPRI mode, 4B/5B must be used.

2.6 Using AIF2 LLD for CPRI fast c&m traffic

In CPRI mode, AIF2 supports Fast Ethernet with 4B/5B encoding over the control words. 4B/5B encoding occurs on top of the SerDes 8B/10B encoding. Ethernet frame format is defined as part of IEEE 802.3. However, it should be noted that 4B/5B encoding follows the big endian convention and transmits the MSB of the second nibble first, which can cause issue with some radio equipment. For details, please check KeyStone-I and KeyStone-II errata advisories. One of the workaround for this limitation is to use Null delimiter option instead for 4B/5B encoding.

AIF2 LLD supports both Null delimiter and 4B/5B encoding options to allow the implementation of the software workarounds presented in the Advisory list. The fast c&m example of AIF2 LLD implements the software workarounds mentioned in the errata usage notes “AIF2 CPRI FastC&M Restrictions and Usage Note”.

3. Programming guidelines

3.1 Driver configuration

3.1.1 AIF_ConfigObj data structure

The AIF2 LLD uses a structure called AIF_ConfigObj. It exposes high level parameters to allow users to configure AIF2 with a general understanding of how AIF2 HW works. The user has to fill in AIF_ConfigObj fields such as the type of antenna traffic, the number of links, the protocol, the data width, the DMA mode etc...

AIF2 LLD is then in charge of translating these parameters into AIF2 FL parameters, and then call the appropriate FL APIs to set AIF2 registers. AIF2 configuration examples are provided in the test examples delivered with the LLD (aif2/test folder).

The high-level or global parameters

These are the parameters that are global for the system such as:

- The AIF2 SerDes input clock

- The operating mode (WCDMA, LTE FDD, LTE TDD, DUAL MODE, GENERIC PACKET...)
- The DMA mode (PKTDma or DIO engine)
- The radio interface protocol (OBSAI, CPRI)
- The AIF2 timer synchronization source (external, software)

Examples of configuration for general parameters

WCDMA test case

Mode: WCDMA
Clock speed: Pick one that fits the Hardware specification
Protocol: For WCDMA, the LLD supports both CPRI and OBSAI protocol
DMA: For WCDMA, the LLD makes use of DIO engine mode
SyncSource: AIF2 LLD support software sync or external rad/phy sync.

LTE test case

Mode: LTE (TDD or FDD)
Clock speed: Pick one that fits the Hardware specification
Protocol: For LTE, the LLD supports both CPRI and OBSAI protocol
DMA: For LTE, the LLD makes use of the PktDMA mode
SyncSource: AIF2 LLD support software sync or external rad/phy sync.

DUAL MODE test case

Mode: DUAL MODE
Clock speed: Pick one that fits the Hardware specification
Protocol: For DUAL MODE, the LLD supports CPRI only for the moment
DMA: For DUAL MODE, the LLD makes use of the DIO mode
SyncSource: AIF2 LLD support software sync or external rad/phy sync.

GENERIC PACKET test case

Mode: GENERIC PACKET
Clock speed: Pick one that fits the Hardware specification
Protocol: For GP, the LLD supports both CPRI and OBSAI protocol
DMA: For GP, the LLD makes use of the PktDMA mode
SyncSource: AIF2 LLD support software sync or external rad/phy sync.

3.2 CPRI and OBSAI link configuration

As part of the high-level parameters, the AIF_ConfigObj data structure contains specific structures, such as 6*AIF_LinkConfigObj, which holds the configuration structure for each antenna link.

AIF2 hardware supports the following features:

- 6 antenna links
- 6 GHz SerDes
- Independent Link rate per link – (exception is CPRI 5x rate, all or none)
- 64 max AxC per link (for WCDMA)
- 8 max AxC per link (for LTE)

The AIF2 LLD link setup structure contains most configurable settings for a given link. The AIF2 LLD supports both 4x and 8x link rates. 2x and 5x are not supported at this stage. Most test cases available as part of the LLD deliveries are 4x link rate tests. It is mandatory for the user to properly set each link parameters for all the enabled links:

- the sample rate
- the data type and width for outbound traffic
- the data type and width for inbound traffic
- the selected dio engine (in case of WCDMA or DUAL MODE)
- the number of AxC for the Tx side (Pe)
 - If not filled it will automatically set to the maximum value in AIF_calcParameters()
- the number of AxC for the Rx side (Pd)
 - if not filled it will automatically set to the maximum value AIF_calcParameters()
- For DUAL MODE, the selected link mode to use (LTE, WCDMA)
- For LTE TDD, the selected UL-DL and Special Subframe configurations

Examples of link configurations

| | Link rate | outbound data type | outbound data width | inbound data type | inbound data width | dio engine | mode |
|------------------------|-----------|--------------------|---------------------|-------------------|--------------------|------------|-------|
| WCDMA | 4x | UL | 8 bits | DL | 16 bits | Dio 0 | NA |
| LTE | 4x | UL | 15 bits | DL | 15 bits | NA | NA |
| DUAL MODE (Wcdma link) | 4x | UL | 8bits | DL | 16 bits | Dio 0 | WCDMA |
| DUAL MODE (Lte Link) | 4x | UL | 15 bits | DL | 15 bits | Dio 1 | LTE |

Those parameters are the mandatory ones for each application using the AIF2 LLD.

In the case of retransmission, the user will need to pass extra parameters in the link configuration data structure, such as timing and position in the daisy chain topology. For this matter, please go to 3.3.2

3.2.1 Antenna container configuration

As a default behavior of AIF2 LLD, AIF_calcParameters() sets the link configuration for the maximum termination of AxCs per link based on the link rate:

It shall be noted that if using a format different than AIF2_LTE_CPRI_1b1, the user has to pay attention to C6670 advisory “AIF2 CPRI LTE Ingress Antenna Carrier Packing Issue” for silicon revision <= 1.0.

3.3 AIF2 physical Timings parameters

Within the AIF2, there are some internal delays in both egress and ingress directions. The AIF2 LLD allows the user to only specify delayTx and delayRx reference timing parameters and, in turn, configure all other internal delays automatically. If those 2 parameters are set to 0, AIF2 LLD will use default parameters that work with most KeyStone-I and KeyStone-II evaluation boards (TI EVM), based on pe2Offset base value. These timings are computed using the AIF_calcParameters() function.

Here are the default values for this offset.

| pe2Offset | WCDMA | LTE |
|-----------|-------|-----|
| OBSAI | 610 | 310 |
| CPRI | 490 | 310 |

The AIF2 LLD programs the internal delay using the following formulas. All these parameters are expressed in byte clocks (80 for OBSAI and 64 for CPRI):

```
if (delayRx==0) delayRx= pe2Offset (default for TI EVM);
```

```
if (delayTx==0) delayTx= pe2Offset (default for TI EVM);
```

```
PE1Offset = pe2Offset - 10;
```

```
if(AIF2FL_LINK_PROTOCOL_OBSAI==hAif->protocol)
```

```
deltaOffset = delayTx + 60 + 80 * nodeTx;
```

```
piMin = delayRx + 60 + 80 * nodeRx;
```

```
if(AIF2FL_LINK_PROTOCOL_CPRI==hAif->protocol)
```

```
deltaOffset = delayTx + 60 + 64 * nodeTx;
```

```
piMin = delayRx + 60 + 64 * nodeRx;
```

```
piMax = piMin + 100;
```

3.3.1 Pi value

If AIF2 is connected with an external device such as a radio head or a CPRI relay solution, an adjustment of Pi value for ingress traffic may be needed, as this external equipment should include a new delay. Pi corresponds to the delay from the physical frame boundary to the actual start of the master frame boundary on the ingress side. It may then include the delay of the RF chain in some cases.

To adjust Pi value on a given system, AIF2 HW features a Pi capture mechanism in its receive mac module (RM) and these captured values can be read from the AT Pi Captured Value Register for a given link.

3.3.2 Using link retransmission (RT)

The link retransmission is handled some specific settings for the link configuration. When doing retransmission on a given link, the modules PD, PE, DIO and DB are not exercised. That means that no AxC will be associated with this link and these modules can be disabled if no other AxC is terminated at that node in the daisy chain.

They are two aspects to consider for the programming of retransmission: the RT module itself, and the specific AIF2 timing for the link.

For the RT module setting part, AIF2 LLD exposes the following two parameters:

- **RtEnabled**
Enables link retransmission. Default value is null.
- **RtLinkRout**
Selects the destination link of the retransmission.

To deal with retransmission timings, AIF2 LLD is making use of the Tx/Rx node position in the antenna daisy chain. Here is a definition of those 2 parameters:

- **nodeTx**
This parameter allows setting the Delta offset on the given link as described in 3.3
A retransmission introduces a delay of ~ 1 Wcdma chip time. That means 80 byte clock for OBSAI and 64 byte clock for CPRI. The nodeTx parameter allows inserting a certain delay based on the place of the current KeyStone SoC in the daisy chain. For a direct communication between two devices, the nodeTx value needs to be set to '0'. For the 1st retransmission node, set it to 1, and so on.
- **nodeRx**
This parameter allows setting the piMin value on the given link as described in 3.3.
A retransmission introduces a delay of ~ 1 Wcdma chip time. That means 80 byte clock for OBSAI and 64 byte clock for CPRI. The nodeRx parameter allows inserting a certain delay based on the place of the current KeyStone SoC in the daisy chain. For a direct communication between two devices, the nodeTx value needs to be set to '0'. For the 1st retransmission node, set it to 1, and so on.

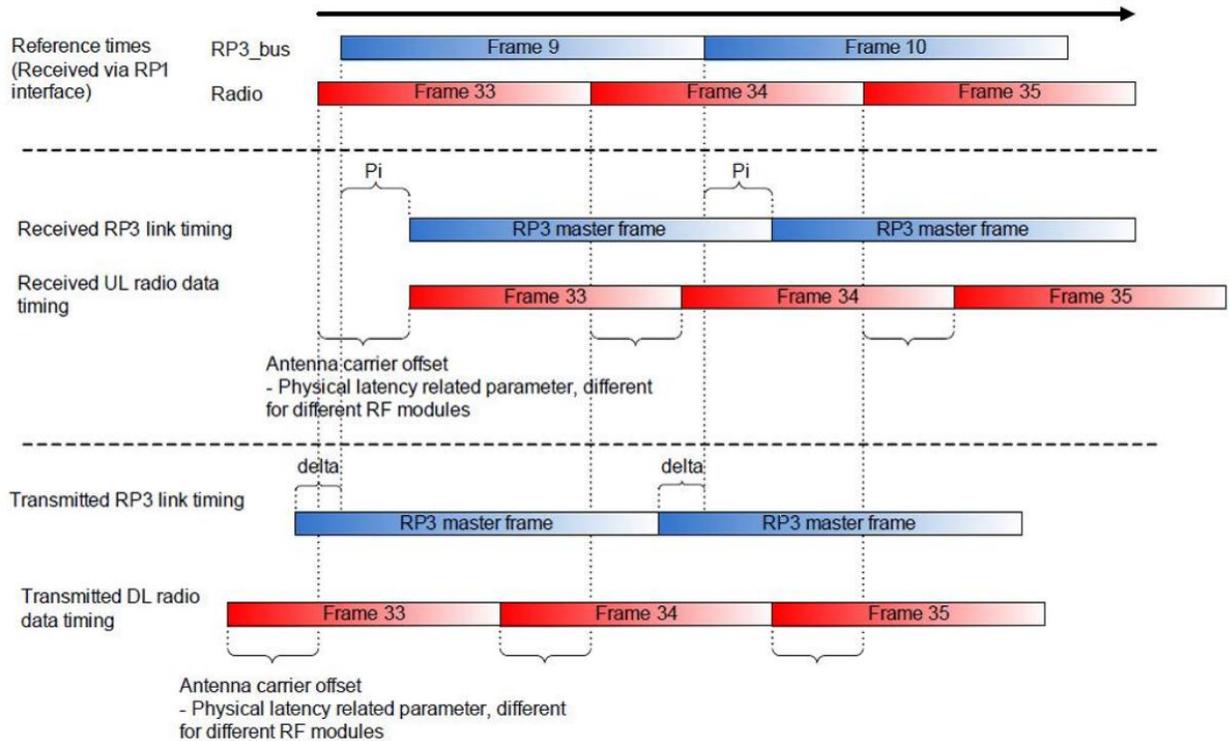


Figure 3: Pi and Delta Timing example

Delta offset is used for transmission, and Pi for reception. So for a communication between 2 devices, the Delta of the one who transmit the data should be aligned with the Pi of the receive one.

Example

- Basic retransmission with link 0 and 2 DSPs:
The purpose here is to send data from DSP 0 over link 0 to DSP 1. The DSP 1's link 0 is set to be retransmitting via itself. DSP 0 receive the data via link 0 and then can send it to the RAC or whatever the application require.

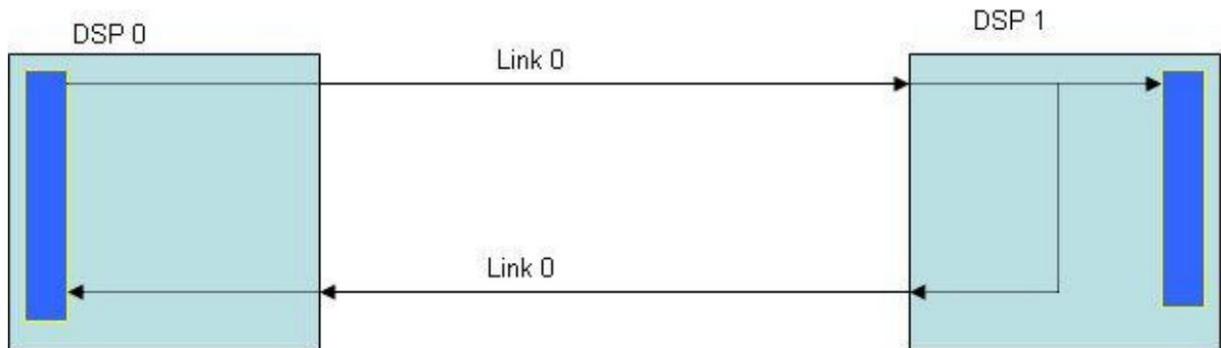


Figure 4: Link forwarding and retransmission between 2 devices

In order to realize that setup, the RT parameters have to be set like that:

| | DSP 0 | DSP 1 |
|------------|-------|--------|
| RtEnabled | 0 | 1 |
| RtLinkRout | NA | link 0 |
| nodeTx | 0 | 0 |
| nodeRx | 1 | 1 |

For DSP1, link 0 is the first to send data (nodeTx=0), so the first to receive these data is the DSP 1 (nodeRx=0). Then for the retransmission via link 0 this is the second Transmission (nodeTx=1) and so the associated reception is the DSP 0's link 0 (nodeRx = 1).

Note: For pure retransmission, the PD and PE modules can be disabled. It can be done by setting `numPeAxC` and `numPdAxC` to 0 after the call to `AIF_calcParameters()` function.

3.4 DIO engines

Direct IO (DIO) means that a peripheral has dedicated custom logic that implements data movement (as opposed to using EDMA or CPU reads/writes). For AIF2, custom circuit is built to handle data movement requirements unique to WCDMA. AIF2 PKTDMA module is making use of channel number 128 for Direct IO purposes. That is why, even if DIO mode is used with AIF2 LLD, calls to both `AIF_initDio()` and `AIF_initPktDma()` is required.

As part of the high-level parameters, the `AIF_ConfigObj` data structure contains specific structures, such as `3* AIF_DioConfigObj`, which holds the configuration structure for each DIO engine.

DIO engines are periodic engines tight to dedicated AIF2 timer events as opposed to PktDMA channels which are driven by data arrival. The AT module generates internal system events that control the DIO engine timing. The AIF2 LLD programs these events with the following periodicity:

- Four chips event for TAC/DL
- Eight chips event for RAC/UL

3.4.1 Ingress and Egress parameters

| AD register fields | Description | LLD implementation |
|----------------------------|---|---|
| <code>num_qw</code> | Number of QuadWord per AxC | Set one QW for DL, two QW for UL |
| <code>num_axc</code> | Number of AxCs associated with the dio engine | Calculated from the number of link(s) associated with the DIO engine |
| <code>dma_base_addr</code> | Vbus source or destination base address | in/out circular buffer from <code>AIF_ConfigObj/AIF_DioConfigObj</code> |
| <code>dma_burst_In</code> | Maximum dma burst length | Default is 4 QW |
| <code>dma_ch_en</code> | DMA channel enable/disable | Set if DIO engine is used |

| | | |
|-----------------------|--|--|
| rsa_en | Data type selection | 1 for UL 0 for DL |
| dma_num_blk | Number of data blocks to transfer before wrapping back to dma_base_addr | inNumBlock/outNumBlock from AIF_ConfigObj/ AIF_DioConfigObj |
| dma_brst_addrs_stride | DMA burst address stride after each DMA burst, the DMA address will increment by this amount | Computed by LLD |
| dma_blk_addrs_stride | DMA block address stride after transferring each dma block (every event time), the DMA address will increment by this amount | Computed by LLD, set to 0x80 if usedWithRAC field in AIF_DioConfigObj is set to match RAC front end format |
| dbcn0 ~ 63 | Match dbcn order to each data buffer channel number | Computed by LLD |

3.4.2 DIO and RAC operation

The ingress DIO provides three DMA engines to control the transfer of data from multiple DB buffers to each of three destinations: RAC, L2/RSA or DDR3.

RAC example

Below is an example of how the DIO is programmed for RAC DMA. Trigger period is 8 chips. DIO ingress is programmed to operate this way if usedWithRAC is set.

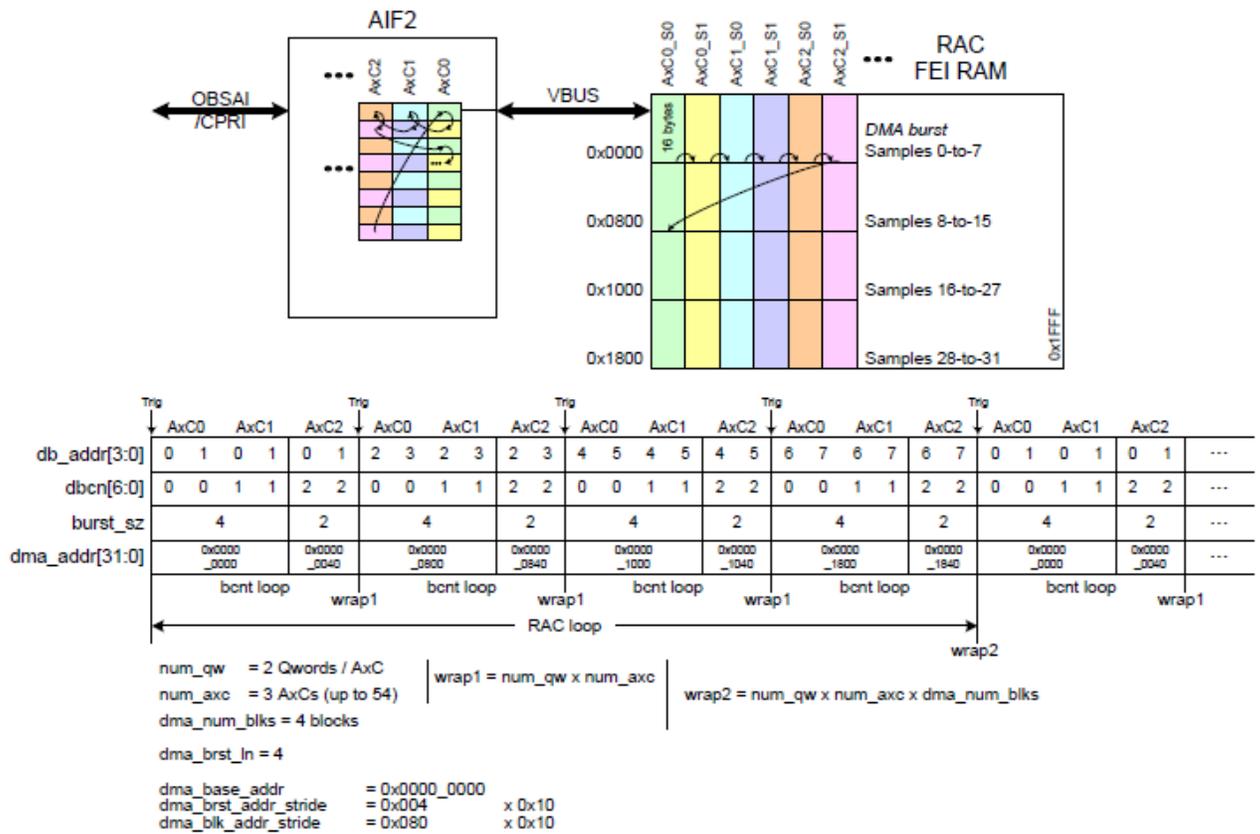


Figure 5: AIF2 to RAC DIO configuration

As the DMA engine traverses the DBCN LUT, it is grouping data transfers into bursts of 4 Qwords. When it only has a burst of two left, it will only do a burst of two. The first burst will start at VBUS address of **dma_base_addr**. After the first burst, the next burst will start at a VBUS address of **dma_base_addr + dma_brst_addr_stride** and so on until the DBCNT has been exhausted. After the next trigger, the DMA engine traverses the DBCNT again with starting address of **dma_base_addr + dma_blk_addr_stride**. The next burst will start at a VBUS address of **dma_base_addr + dma_blk_addr_stride + dma_brst_addr_stride** and so on until the DBCNT has been exhausted. The DMA engine will proceed in this manner until **dma_num_blks** have been transferred and then start again from the beginning.

3.4.3 DIO and TAC operation

The egress DIO provides three DMA engines to control the transfer of data from each of the three destinations, TAC, L2/RSA or DDR3 to multiple DB buffers. Precise timing of transfers occurs based on AT signaling.

TAC example

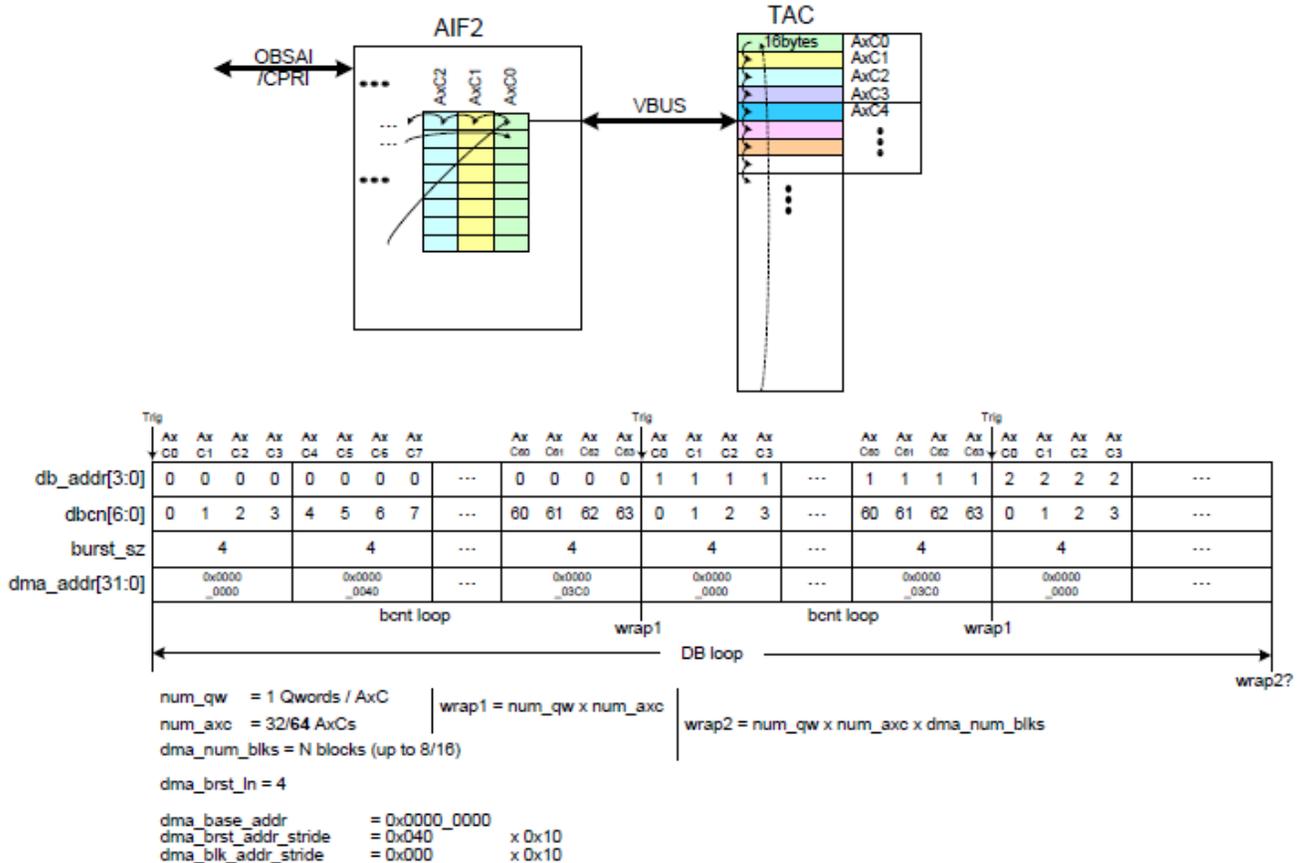


Figure 6: TAC to AIF2 DIO configuration

See Ingress example.

3.4.4 DIO parameters

`aifObj.linkConfig[i].dioEngine = (0,1 or 2)`

This parameter associates a DIO engine to a given link. Then the mapping between AxC, link and DIO is made by the LLD.

The basic DIO parameters that have to be set by the user are:

- `aifObj.dioConfig[i].out`
Buffer start address for this DIO engine for egress.
- `aifObj.dioConfig[i].in`
Buffer start address for this DIO engine for ingress.
- `aifObj.dioConfig[i].outNumBlock`
Number of DMA blocks (wrap2) for this DIO engine in egress direction
- `aifObj.dioConfig[i].inNumBlock`
Number of DMA blocks (wrap2) for this DIO engine in ingress direction
- `aifObj.dioConfig[i].usedWithRAC`
When in Wcdma mode, tells whether this DIO engine is used for RAC on ingress side
- `aifObj.dioConfig[i].mode`
When in mix mode, tells whether this DIO engine is used for LTE or WCDMA traffic.
- `aifObj.dioConfig[i].sampleRate`
When in mix mode, a DIO engine can have different timing depending on the traffic it needs to carry. This field allows the user to specify at which rate the DIO needs to pace transfers.

`AIF_initDio()` function must be called prior to `AIF_initHw()` to initialize some of the DIO parameters in AIF2 LLD.

3.4.5 LLD restriction

One DIO engine can handle up to 64 AxC. That means only 4 links (16 AxCs per link) at 4x rate, or 2 links at 8x (32 AxCs per link) can be assigned to the same DIO engine. This is hardware restriction.

AIF2 LLD adds a new restriction: the links associated to a given DIO have to be contiguous. Valid configuration example: link[0:1] on engine0, link[2] on engine1, link[3] on engine2 Invalid configuration example: link[0,2] on engine0, link[1] on engine1, link[3] on engine2 The main motivation for this restriction is the assignment of the AIF2 DB channels across the multiple links and DIO engines.

3.4.6 DIO duplicate ingress traffic on DIO 2

The principle is to set DIO engine 2 as an image of another (DIO0 or DIO 1) that duplicate data to another DIO circular buffer on the KeyStone SoC. It could be, for instance, to another RAC, or to some

DSP memory for debug purposes. The same DBs are allocated to both DIO, so the same data pass through both DIO to two different locations.

- `aifObj.dioConfig[i].duplicateOnDioEng2`
Set to 1 means that DIO engine 2 is dedicated to debug mode and will copy this current DIO.

AIF2 LLD always use DIO engine 2 to duplicate the current DIO. So if DIO duplicate feature is used the application cannot use the DIO engine 2 for traffic purpose. Only one DIO can be debugged at a time, and it can't be DIO engine 2.

- `aifObj.dioConfig[2].in`
Set the buffer destination start address for DIO engine 2 .

3.5 PKTDMA channels

Multicore Navigator is a methodology and a series of hardware accelerator modules that allow the DSP cores and peripherals to effectively transfer packets. It is a safe and managed way that memory can be used to pass data.

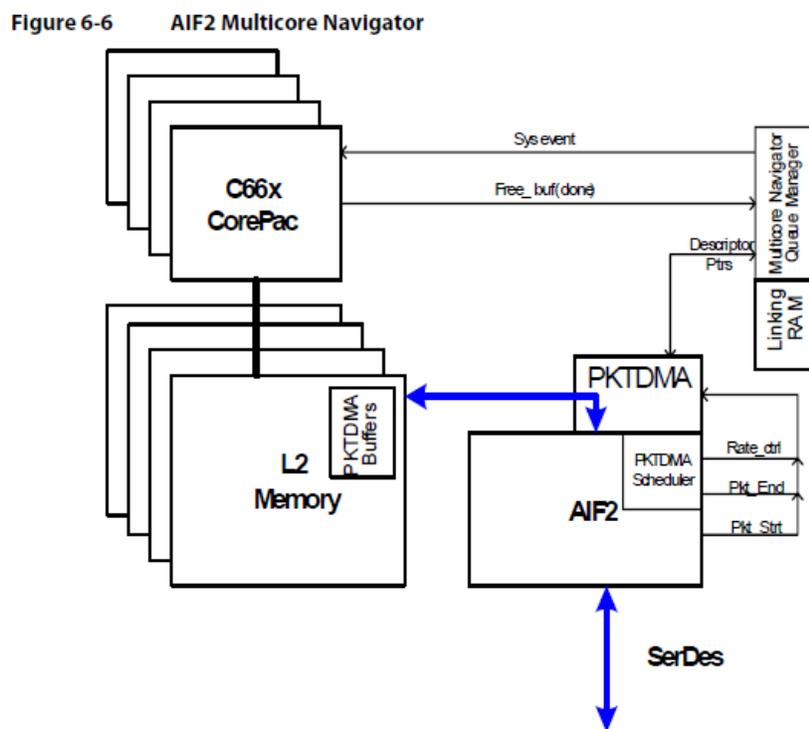


Figure 7: Packet DMA and AIF2

The AIF2 LLD makes use of the QMSS and CPPI LLD to initialize those 2 modules used to enable packet DMA channels and transfers. The user should note that **it is optional to use the AIF2 LLD APIs related to packet DMA channel and hardware queue**. The user application can configure those modules independently from the LLD.

3.5.1 Memory region

The first thing to be done when using the PKTDMA is to define a memory region (QMSS). This memory region will store all the descriptors needed by the application. They are 2 important keys that must be known when creating a memory region:

- First the descriptor number of the memory region must be a power of 2.
- A memory region can support only one buffer size. Meaning that the buffer size must be chosen in advance to handle the largest possible packet size. Example: for a 20 MHz LTE application, the buffer must store the largest LTE symbol, that is the first (2208 samples).

This section must be done by the user as it is application specific.

3.5.2 Antenna containers

The AIF2 LLD provides support to the user in order to set up a PKTDMA transfer via the AIF queues. The picture below describes what is actually done by the LLD software:

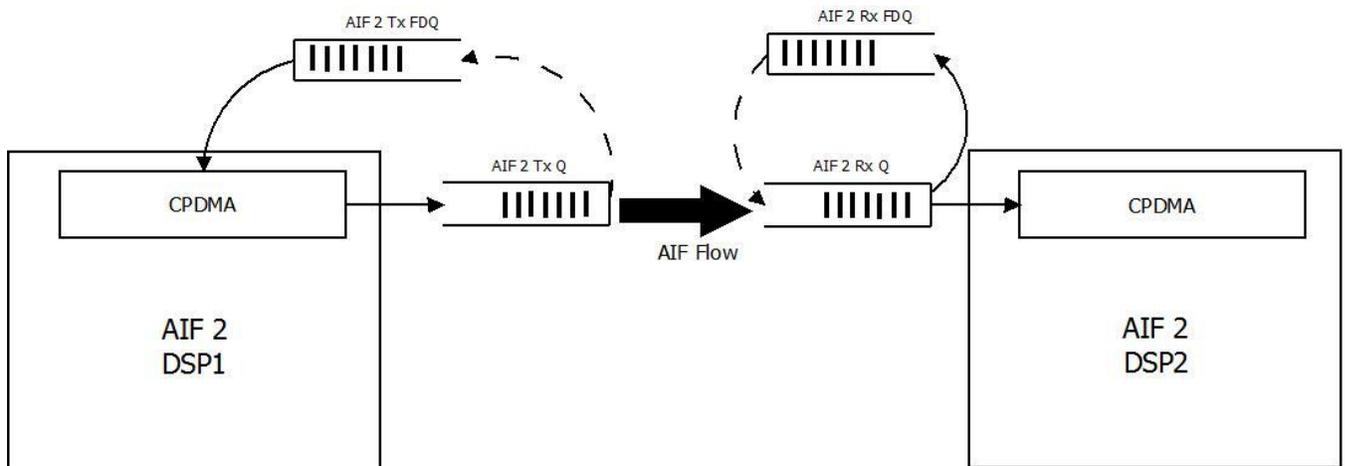


Figure 8: Packet DMA queues (Q) and free descriptor queues (FDQ)

This is a basic AIF flow. The AIF2 LLD will configure this flow if the pktDMA structure of the aifObj structure is set. This structure contains the parameters for the PKTDMA and QMSS setup:

The Ctrl Queues are used for CPRI Fast C&M control words and will be explained later. The AxC Queues are used for AIF traffic. The AIF2 LLD assumes that for each AxC there is a flow. They are 128 pktDMA channels. The last one is for DIO engine, between 124 and 127 this is used for control words. That let 124 channels for AIF AxC.

The PKTDMA and QMSS need to configure:

- A Cppi Handle
- A Tx free descriptor queue
- A Tx Queue
- A Rx free description queue
- A Rx Queue

- A Tx channel
- A Rx channel
- A Rx Flow.

CPPI

This is done by software and nothing has to be done by the user.

Tx free descriptor queue

In order to feed the TxFDQ, there is some information that the user must inform

- `aifObj.pktDmaConfig.txRegionAxC[chan] = Qmss_MemRegion_MEMORY_REGION0;`
This will set all the descriptor of this queue to be taken from the memory region that has been previously configured.
- `aifObj.pktDmaConfig.txNumDescAxC[chan] = NBSYMBOL*2; // double num of Pkts`
Set the numbers of descriptor that is needed to be implementing in the TxFDQ. It should be big enough so that the application never runs out of descriptors.
- `aifObj.pktDmaConfig.txDescSizeAxC[chan] = LTESYMBOLSIZE;`
Set the size of the payload that is attached to the descriptor, as it is monolithic descriptors for AIF PKTDMA. For LTE, set the size of the biggest one (1st symbol).

The queue will be selected by the LLD between all the available generic purpose queue the number will be store in `aifObj.pktDmaConfig.txFqAxC`.

Tx Queue

The Tx Queue is defined by the LLD. It will take the first AIF queue available. There are 128 AIF queues, from queue 512 to 639. The information regarding the queue numbers will be store in `aifObj.pktDmaConfig.txQAxC`.

Rx free descriptor queue

Unlike the TxFDQ and TxQ, the queue wanted for the RxFDQ must be specified. It is mandatory to choose a general purpose queue, and an available one. That is a queue from 896 to 8191.

Then it goes the same as for the configuration of TxFDQ:

- `aifObj.pktDmaConfig.rxRegionAxC[chan] = Qmss_MemRegion_MEMORY_REGION0;`
- `aifObj.pktDmaConfig.rxNumDescAxC[chan] = NBSYMBOL*2; // double num of Pkts`
- `aifObj.pktDmaConfig.rxDescSizeAxC[chan] = LTESYMBOLSIZE*NBCHANNELPERLINK;`
For the descriptor size, you must remember that if you are using the super packet workaround, the receive path will merge the AxC that are transmitted so the descriptor size will be bigger depending on how many AxC you have per link.

Rx Queue

The Rx queue is set be the user. You must choose a generic purpose queue among the entire available one that are from queue 896 to 8191. The LLD will take the number you gave when setting the value of the associated flow `rxFlow[link][AxC].rx_dest_qnum`.

Tx, Rx channel

This is directly set up by the LLD. It will automatically set the channel to match the flow for a given AxC.

Flow

The flow is the way to make the Tx and Rx queues to exchange data. It matches with the channel automatically. Left to configure is some information regarding the Rx recycling method and the destination queue number (RxQ).

- `rxFlow[link][AxC].rx_dest_qnum = MONO_RX_Q+chan;`
Here you must set the chosen number for the RxQ for the given AxC. This allows the transfer to the RxQ of the descriptor that is pushed to TxQ.
- `rxFlow[link][AxC].rx_fdq0_sz0_qnum = MONO_RX_FDQ+chan;`
- `rxFlow[link][AxC].rx_fdq1_qnum = MONO_RX_FDQ+chan;`
- `rxFlow[link][AxC].rx_fdq2_qnum = MONO_RX_FDQ+chan;`
- `rxFlow[link][AxC].rx_fdq3_qnum = MONO_RX_FDQ+chan;`
Configure the RxFDQ in which the descriptor must return after being popping from RxQ.
- `rxFlow[link][AxC].rx_desc_type = (Uint8)Cppi_DescType_MONOLITHIC; // MONO`
For AIF LTE traffic you must always choose Monolithic descriptor type.
- `rxFlow[link][AxC].rx_sop_offset = 12+4;`
For monolithic packet the payload is attached to the descriptor so it is mandatory to inform the size of the Header (12 bytes) + PS (4 bytes). Then the descriptor now that the data is located 16 bytes after the header.

```
aifObj.pktDmaConfig.hRxFlowAxC[chan] = &rxFlow[i][idx];  
aifObj.pktDmaConfig.hRxFlowCtrl[chan] = NULL;
```

3.5.3 Pushing and recycling packets

The AIF2 LLD aims at configure the AIF in order to be able to do LTE, WCDMA and Generic packet transfer. For LTE when using the Multicore Navigator, the application will need to pop and push packet for both sending packet and recycling the ones that are received. This is not done by the LLD has it depends on the application. You can still have a look to the examples that are included in this package who show some transfer that are done at different time base. We add for LTE FDD two AIF events that will help you push and pop at a sub-frame rate (1 ms) or at a slot rate (0.5 ms):

- Sub-Frame: AIF event 6
- Slot time: AIF event 5

3.5.4 CPRI Fast C&M control words

Users should be aware of the required SW workarounds for CPRI fast c&m restrictions. Concerning the PKTDMA traffic for fast c&m, it can be considered as asynchronous to the IQ sample traffic, in the sense that pushing packets on the Tx direction can occur at any time. The AIF2 LLD configures CPRI control stream #0 as the fast c&m one. On reception of fast c&m packets, the application pops the packet descriptor from the associate CPRI control stream #0. Dedicated structure fields (Tx control queue, Rx

control queue, ...) for CPRI control streams are available for configuring and using Cpri fast c&m functionality in aifObj.pktDmaConfig.

3.6 Physical and Radio timers

3.6.1 Using pre-defined AT events and counters

AIF LLD provides pre-configured events to help the user pace its application. Depending on the chosen standard, the pre-defined AT events are set accordingly.

General Event: Available for all the configuration

AT event 7: this event is dedicated to generate the 10 ms frame boundary. This event goes for every application.

This is one of the most important counters as it gives a timing reference. For WCDMA, we use this counter to stop the application after the desired amount of time spent.

AIF2LLD implement a counter base on that timer, named “aifSyncEventCount[1]”. It counts the number of frame that has been sent since the start of the AIF.

WCDMA event

AT event 8: event generating a 4 chip trigger for TAC. Event 8 is specified for this purpose.

AT event 9: event generating a 32 chip trigger for RAC_A. Event 9 is specified for this purpose.

AT event 10: event generating 32 chip trigger for RAC_B. Event 10 is specified for this purpose

For KeyStone-II devices only:

AT event 11: event generating 32 chip trigger for RAC_C. Event 10 is specified for this purpose

AT event 12: event generating 32 chip trigger for RAC_D. Event 10 is specified for this purpose

LTE FDD event

AT event 5: event generating a 0.5 ms timeslot. User can use this interruption to pop and push packet to feed the AIF2 at a slot time basis (7 symbols per slot, which correspond to 7 packets pushed or popped).

AT event 6: event generating a 1 ms sub-frames time. User can use this interruption to pop and push packet to feed the AIF2 at a slot time basis (14 symbols per slot, which correspond to 14 packets pushed or popped).

AT event 0: This is the event that trigs the EDMA3 for the Superpacket workaround. For TNYquist Rev1.0.

LTE TDD event

AT event 5: event generating a 0.5 ms timeslot. User can use this interruption to pop and push packet to feed the AIF2 at a slot time basis (7 symbols per slot, which correspond to 7 packets pushed or popped).

AT event 6: event generating at the granularity of a symbol. User should use this interruption to pop and push packet to feed AIF2 at a slot time basis.

DUAL MODE event

AT event 5: event generating a 0.5 ms timeslot. In dual mode, you will need to use this interruption to recycle

AT event 6: event generating at the granularity of a symbol. User should use this interruption to recycle the packet made for the re-packetization of LTE traffic and then push it to the FFTC queues.

3.6.2 Adding application-specific AT events

AIFLLD aims at simplifying the ad of AT event for application specific purpose. Just be aware to not configure an Event that is already used by the LLD, unless changes that implies for the application are known.

To add an event, first declare a new Aif2AtEvent structure. Then, configure all the parameters based on the needs:

EventSelect: select the event used for this AT. Remember to use a free one, from 0 to 6.

EventOffset: the AT event will start AT EvtStrobeSel + EventOffset. In order for that to work, set a value that is inferior at the Event modulo, otherwise it will never trigger.

EvtStrobeSel: The event can be triggered at different source time base that have been configure by the LLD:

- The AIF2FL_RADT_FRAME: basically at a frame boundaries
- AIF2FL_RADT_SYMBOL: trigger every symbol time (the shortest available, in case that strobe is selected the event will trig every symbol, the configuration of modulo is not needed as the strobesel wrap back the event counter)
- AIF2FL_PHYT_FRAME: at the PHY frame boundary if it is different from Radt.
- AIF2FL_ULRADT_FRAME: based on UL configuration. By default no recommended.
- AIF2FL_DLRADT_FRAME: based on DL configuration. (Not implemented in AIF2LLD).

EventModulo: modulo sets the frequency of the AT event. It is measure in CPRI or OBSAI byte clock. So for a 4 chip AT event, the correct value is $4*80-1$ for OBSAI and $4*64-1$ for CPRI.

EventMaskLsb: set 0xFFFFFFFF; this is only for GSM (not supported in AIF2 LLD)

EventMaskMsb: set 0xFFFFFFFF; this is only for GSM (not supported in AIF2 LLD)

Now that the event is configure, call the **AIF_addAtEvent(Aif2Fl_AtEvent* hAtEvent)** function between AIF_initHw(&aifObj) and AIF_startHw(&aifObj).

Example

This example set an AT event that will trig every 256 chip on the event 6, generated from the RADT clock.

Implement the interruption generated by event 6.

```
void chipBoundaryIsr_256() {  
    Application code  
}
```

In the Main:

```
AIF_initHw(&aifObj);  
// add a 256-chip event based on RadioTimer event 6  
atEvt6.EventSelect = AIF2FL_EVENT_6;  
atEvt6.EventOffset = 0;  
atEvt6.EvtStrobeSel = AIF2FL_RADT_FRAME; // frame strobe select.  
atEvt6.EventModulo = (256*64) - 1; // 256-chip modulo expressed in CPRI byte clock  
atEvt6.EventMaskLsb = 0xFFFFFFFF;  
atEvt6.EventMaskMsb = 0xFFFFFFFF;  
AIF_addAtEvent(&atEvt6);  
// add user routine on event 6  
aif2evt6_userIsr = chipBoundaryIsr_256;  
AIF_startHw(&aifObj);
```

3.6.3 Triggering physical and radio timers

Tbd

3.7 AIF2 exceptions

The AIF2 had a large number of system error/alarm condition signal. The function of the Exception Interrupt Handler (EE) is to aggregate this large number of errors/alarm condition signals to a number of interrupts that can be used to generated DSP interrupts.

The AIF2 LLD offers the possibility to enable the AIF2 exception handler. The LLD just trace the most significant signal of interest, and offers the possibility to watch them as they are reported in the aifObj, in the AIF_EeCountObj structure.

In order to activate the EE handler, the AIF_enableException(&aifObj) function must be call some frame after the AIF start its timers. It will then store all the exception spotted in the aifObj if there is a malfunction, and raising the flag of EE.

Before the reset of the AIF2, the interruption that capture the EE must be disable be calling UTILS_aif2ExceptIntDisable function, otherwise it may capture some EE that are due to the reset of the SerDes.

Interruption: TBD

4. AIF2 hardware initialization

4.1 Sequence of AIF2 LLD calls at start-up

Here is the sequence that is expected by AIF2 LLD for proper initialization. Upon the execution of this sequence, the AIF2 hardware is expecting a frame sync trigger (Software, radsync/physync). The AIF2 LLD test utilities have support for that procedure for both loopback and dual-DSP setups.

First it calculates the timing of the application, then it configures the DIO (disable for LTE, enable with a certain amount of AxC for WCDMA), after that it initializes the QMSS and CPPI (CW for WCDMA, CW and channel for LTE and GENERIC PACKET). Finally it sets the AIF2 based on the information above. But it only sets a pointer on a FL object, to set the register of the AIF it needs to call the start hardware function that will configure the AIF2. It is possible to change some register before calling the start hardware function for special purpose such as adding a new AT event.

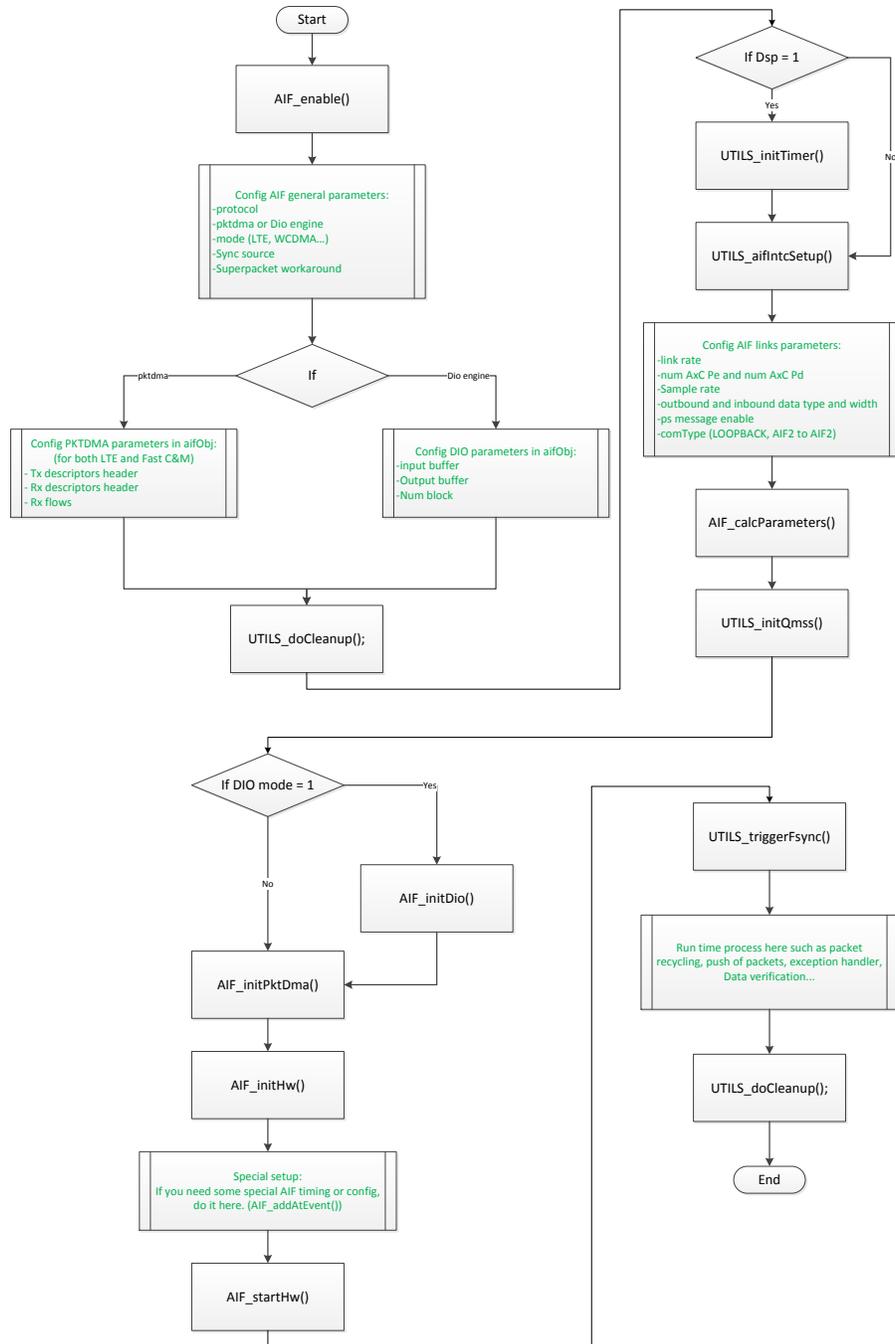


Figure 9: AIF2 driver initialization sequence

4.2 AT event programming at run-time

This section will be added in a later release of the driver.

4.3 Re-initialization

The AIF2 LLD provides a way to implement a clean-up sequence for the AIF2 Hardware. **AIF_resetFsync()** and **AIF_resetAif()** are Sw-resetting the AIF2 hardware and LLD objects. The AIF2 LLD test utility, **UTILS_doCleanup()** implements a full cleanup sequence in the LLD tests. It should be noted though that for the queue manager sub-system (QMSS), the linking RAM or memory regions cannot be re-configured without a "System Reset".

5. AIF2 LLD testing

The directory [your path]/aif2/test contains the AIF2 LLD test suite and some CSL utilities. Those can be used as examples for the LLD user. See the release notes for the exhaustive list of available tests.

6. AIF2 LLD example configuration

Based on AIF2LLD examples that can be found within the LLD package, the purpose of this chapter is to describe the configuration that is being made for some of these tests.

6.1 testAIF2_Wcdma4x

This test is a generic test for WCDMA. It allows traffic for CPRI or OBSAI, for one or multiple links on one or multiple DIO engines.

This section describes the test for one link one DIO, and will explain how the LLD configure the DIO engine by default.

General parameters

We want to make a WCDMA test case, with CPRI protocol. The Antenna Interface will use the DIO engine as the DMA solution.

| | |
|---------------------|---------------------------|
| protocol | AIF2FL_LINK_PROTOCOL_CPRI |
| pktdmaOrDioEngine | AIF2FL_DIO |
| mode | AIF_WCDMA_MODE |
| aif2TimerSyncSource | AIF2FL_CHIP_INPUT_SYNC |

Configuration DIO and data buffers

We assume that for this test, we need only one DIO engine and only one link. We will not use the full capacity of the link that is 15 AxCs for CPRI, but only 2 AxCs to make the description clearer. The tests are made for LOOPBACK, so the data are the same type on Rx and Tx (UL or DL).

DL configuration

Dio parameters for DL, defined by user:

```
aifObj.dioConfig[i].out = UTILS_local2GlobalAddr(&dio_data[i][0]);
```

```

aifObj.dioConfig[i].in      = UTILS_local2GlobalAddr(&dio_result[i][0]);
aifObj.dioConfig[i].inNumBlock = 8;
aifObj.dioConfig[i].outNumBlock = 8;

```

Link parameters to only enabled 2 AxCs for the link:

```

aifObj.linkConfig[i].numPeAxC = 2
aifObj.linkConfig[i].numPdAxC = 2

```

The AIF2 LLD will configure the DIO engine with these values by default:

| | Ingress | Egress |
|---------------------------|-----------------|-----------------|
| NumAxC | 2 (-1) | 2 (-1) |
| DmaNumBlock | 8 (-1) | 8 (-1) |
| DmaBaseAddr | dio_result | dio_data |
| NumQuadWord | AIF2FL_AD_1QUAD | AIF2FL_AD_1QUAD |
| bEnEgressRsaFormat | FALSE | FALSE |
| bEnDmaChannel | TRUE | TRUE |
| DmaBurstLen | AIF2FL_AD_4QUAD | AIF2FL_AD_4QUAD |
| DmaBlockAddrStride | NumAxC (QW) | NumAxC (QW) |
| DmaBurstAddrStride | 4 (QW) | 4 (QW) |

Based on this value, we can determinate the wrap 1 and wrap 2 of the DIO engine, and by doing so the data buffer size.

- wrap 1 = NumAxC * NumQuadWord
wrap 1 = 2 * 1 = 2 QuadWord = 2*16 = 32 bytes.
- Wrap 2 = wrap 1 * DmaNumBlock
Wrap 2 = 32 * 8 = 256 bytes.
This corresponds to the data buffer size. Once the DIO engine will send 256 bytes of data it will wrap back to the start address of the dio_data buffer.

dio_data

| | 0 | 4 | 8 | C |
|------------|------|------|------|------|
| 0x10800000 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800010 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x10800020 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800030 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x10800040 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800050 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x10800060 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800070 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x10800080 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800090 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x108000A0 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x108000B0 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x108000C0 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x108000D0 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x108000E0 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x108000F0 | AxC1 | AxC1 | AxC1 | AxC1 |

} Wrap 1

} Wrap 2

Figure 10: Wcdma egress DIO data in SoC memory

With this configuration of the DIO engine, a buffer is sent with multiple AxCs interleave.

On the Rx side, as the DIO engine in ingress is setup in the exact same way, it will reconstruct the packet identically.

UL configuration

Dio parameters for UL, defined by user:

```

aifObj.dioConfig[i].out      = UTILS_local2GlobalAddr(&dio_data[i][0]);
aifObj.dioConfig[i].in      = UTILS_local2GlobalAddr(&dio_result[i][0]);
aifObj.dioConfig[i].inNumBlock = 4;
aifObj.dioConfig[i].outNumBlock = 4;
  
```

Link parameters to only enabled 2 AxCs for the link:

```

aifObj.linkConfig[i].numPeAxC = 2
aifObj.linkConfig[i].numPdAxC = 2
  
```

The AIF2 LLD will configure the DIO engine with these values by default:

| | Ingress | Egress |
|--------------------|------------|----------|
| NumAxC | 2 (-1) | 2 (-1) |
| DmaNumBlock | 4 (-1) | 4 (-1) |
| DmaBaseAddr | dio_result | dio_data |

| | | |
|---------------------------|-----------------|-----------------|
| NumQuadWord | AIF2FL_AD_2QUAD | AIF2FL_AD_2QUAD |
| bEnEgressRsaFormat | TRUE | TRUE |
| bEnDmaChannel | TRUE | TRUE |
| DmaBurstLen | AIF2FL_AD_4QUAD | AIF2FL_AD_4QUAD |
| DmaBlockAddrStride | NumAxC*2 (QW) | NumAxC*2 (QW) |
| DmaBurstAddrStride | 4 (QW) | 4 (QW) |

Based on this value, we can determinate the wrap 1 and wrap 2 of the DIO engine, and by doing so the data buffer size.

- wrap 1 = NumAxC * NumQuadWord
wrap 1 = 2 * 2 = 4 QuadWord = 4*16 = 64 bytes.

- Wrap 2 = wrap 1 * DmaNumBlock
Wrap 2 = 64 * 4 = 256 bytes.

This corresponds to the data buffer size. Once the DIO engine will send 256 bytes of data it will wrap back to the start address of the dio_data buffer.

dio_data

| | 0 | 4 | 8 | C |
|------------|------|------|------|------|
| 0x10800000 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800010 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800020 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x10800030 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x10800040 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800050 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800060 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x10800070 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x10800080 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x10800090 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x108000A0 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x108000B0 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x108000C0 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x108000D0 | AxC0 | AxC0 | AxC0 | AxC0 |
| 0x108000E0 | AxC1 | AxC1 | AxC1 | AxC1 |
| 0x108000F0 | AxC1 | AxC1 | AxC1 | AxC1 |

Figure 11: Wcdma ingress DIO data in SoC memory

With this configuration of the DIO engine, a buffer is sent with multiple AxCs interleave.

On the Rx side, as the DIO engine in ingress is setup in the exact same way, it will reconstruct the packet identically.

6.2 testAIF2_cpriWcdmaCheckRF

This is the test that has been created in order to test the CPRI RELAY setup. It has a special configuration for the DIO engine as we want to send 2 different signals on AxCO and AxC1. For that purpose we bypass the default configuration of the DIO and created one that separated the data for AxCO and AxC1 into 2 buffers.

General parameters

We want to make a WCDMA test case, with CPRI protocol. The Antenna Interface will use the DIO engine as the DMA solution.

| | |
|---------------------|---------------------------|
| protocol | AIF2FL_LINK_PROTOCOL_CPRI |
| pktdmaOrDioEngine | AIF2FL_DIO |
| mode | AIF_WCDMA_MODE |
| aif2TimerSyncSource | AIF2FL_CHIP_INPUT_SYNC |

Configuration DIO and data buffers

For this test we are using only one link, and 2 AxCs for this link.

The data sent is DL format, while the one that is received from the RF is UL.

```

aifObj.dioConfig[0].out      = UTILS_local2GlobalAddr(&dio_data[0][0]);
aifObj.dioConfig[0].in       = UTILS_local2GlobalAddr(&dio_result[0][0]);
aifObj.dioConfig[0].outNumBlock = 2400;
aifObj.dioConfig[0].inNumBlock = 1200;

```

After the call of AIF_initHw(&aifObj) function we reconfigure the DIO to change the default parameters:

| | Ingress | Egress |
|---------------------------|-----------------|-----------------|
| NumAxC | 2 (-1) | 2 (-1) |
| DmaNumBlock | 1200 (-1) | 2400 (-1) |
| DmaBaseAddr | dio_result | dio_data |
| NumQuadWord | AIF2FL_AD_2QUAD | AIF2FL_AD_1QUAD |
| bEnEgressRsaFormat | TRUE | FALSE |
| bEnDmaChannel | TRUE | TRUE |
| DmaBurstLen | AIF2FL_AD_2QUAD | AIF2FL_AD_1QUAD |
| DmaBlockAddrStride | 2 (QW) | 1 (QW) |
| DmaBurstAddrStride | 2400 (QW) | 2400 (QW) |

Based on this value, we can determinate the wrap 1 and wrap 2 of the DIO engine, and by doing so the data buffer size.

Egress DIO config:

- wrap 1 = NumAxC * NumQuadWord
wrap 1 = 2 * 1 = 2 QuadWord = 2*16 = 32 bytes.
- Wrap 2 = wrap 1 * DmaNumBlock
Wrap 2 = 32 * 2400 = 76800 bytes.

This corresponds to the data buffer size. Once the DIO engine will send 76800 bytes of data it will wrap back to the start address of the dio_data buffer.

| | 0 | 4 | 8 | C | |
|------------|------|------|------|------|------------|
| 0x10800000 | AxC0 | AxC0 | AxC0 | AxC0 | block 1 |
| 0x10800010 | AxC0 | AxC0 | AxC0 | AxC0 | block 2 |
| 0x10800020 | AxC0 | AxC0 | AxC0 | AxC0 | block 3 |
| 0x10800030 | AxC0 | AxC0 | AxC0 | AxC0 | block 4 |
| ... | ... | ... | ... | ... | |
| 0x108095E0 | AxC0 | AxC0 | AxC0 | AxC0 | block 2399 |
| 0x108095F0 | AxC0 | AxC0 | AxC0 | AxC0 | block 2400 |
| 0x10809600 | AxC1 | AxC1 | AxC1 | AxC1 | block 1 |
| 0x10809610 | AxC1 | AxC1 | AxC1 | AxC1 | block 2 |
| 0x10809620 | AxC1 | AxC1 | AxC1 | AxC1 | block 3 |
| 0x10809630 | AxC1 | AxC1 | AxC1 | AxC1 | block 4 |
| ... | ... | ... | ... | ... | |
| 0x10812BE0 | AxC1 | AxC1 | AxC1 | AxC1 | block 2399 |
| 0x10812BF0 | AxC1 | AxC1 | AxC1 | AxC1 | block 2400 |

Wrap 1 Wrap 2

Figure 12: Wcdma egress DIO data in SoC memory

That configuration allows separating the data for each AxCs into dedicated buffers. It is then easy to check the data corresponding to a specific AxC.

Ingress DIO config:

- wrap 1 = NumAxC * NumQuadWord
wrap 1 = 2 * 2 = 4 QuadWord = 4*16 = 64 bytes.
- Wrap 2 = wrap 1 * DmaNumBlock
Wrap 2 = 64 * 1200 = 76800 bytes.

In order to have the same buffer size as egress we divide the block num by 2.

| | 0 | 4 | 8 | C | |
|------------|------|------|------|------|------------|
| 0x10800000 | AxC0 | AxC0 | AxC0 | AxC0 | block 1 |
| 0x10800010 | AxC0 | AxC0 | AxC0 | AxC0 | block 1 |
| 0x10800020 | AxC0 | AxC0 | AxC0 | AxC0 | block 2 |
| 0x10800030 | AxC0 | AxC0 | AxC0 | AxC0 | block 2 |
| ... | ... | ... | ... | ... | |
| 0x108095E0 | AxC0 | AxC0 | AxC0 | AxC0 | block 1200 |
| 0x108095F0 | AxC0 | AxC0 | AxC0 | AxC0 | block 1200 |
| 0x10809600 | AxC1 | AxC1 | AxC1 | AxC1 | block 1 |
| 0x10809610 | AxC1 | AxC1 | AxC1 | AxC1 | block 1 |
| 0x10809620 | AxC1 | AxC1 | AxC1 | AxC1 | block 2 |
| 0x10809630 | AxC1 | AxC1 | AxC1 | AxC1 | block 2 |
| ... | ... | ... | ... | ... | |
| 0x10812BE0 | AxC1 | AxC1 | AxC1 | AxC1 | block 1200 |
| 0x10812BF0 | AxC1 | AxC1 | AxC1 | AxC1 | block 1200 |

Wrap 1 Wrap 2

Figure 13: Wcdma ingress DIO data in SoC memory

6.3 testAIF2_cpriRAC

Tbd

6.4 testAIF2_cpriDILte

This is the reference test for LTE data traffic. This chapter will focus on the PKTDma configuration to explain the LLD requirement.

Memory configuration

For LTE traffic, the LLD makes use of the monolithic descriptors. In order to make it works, memory region with the right number of descriptor and the proper size has to be defined and configured.

For LTE 20 MHz, a symbol size is equal to:

- 8848 Bytes for the First symbol of a slot (cyclic prefix different)
- 8784 for the 6 others symbol.

In this test, we are using the predefined AT event 6 that is sub-frame based. There are 14 symbols in one sub-frame. We will need some sort of ping-pong between the descriptor, so that means that we need 14 * 2 descriptors for TX side, and 14 * 2 descriptors for Rx side. A total of 56 descriptors per AxC.

7. Different AxC sampling rate on a same CPRI link

7.1 Dual LTE

In order to support multiple LTE sampling rates on a single link, the LLD uses the AxC level parameters to properly configure the AIF2 protocol encoder and decoder modules.

As part of the AIF_LinkConfigObj properties, LLD users can set multiRate field to 1. When this mode is set, the LLD takes the sample rate for each AxC instead of the sample rate of the link for every AxCs mapped on it. Therefore the user application must set the sampleRate and cpriPackMode for EVERY enabled AxCs. All these parameters are found into aifObj.AxCconfig[[]].

For backward compatibility, the AIF2 LLD assumes that multiRate mode is not enabled by default.

Dual LTE configuration example

Consider a dual carrier system that needs to configure AIF2 for 2xLTE20+2xLTE10 on CPRI 8x link index 0. Then the LLD instance requires the following parameter settings:

```
hAif->linkConfig[0].numPeAxC          = 4
hAif->linkConfig[0].numPdAxC          = 4
hAif->linkConfig[0].linkEnable        = 1
hAif->linkConfig[0].linkRate          = AIF2FL_LINK_RATE_8x
hAif->linkConfig[0].outboundDataType  = DATA_TYPE_DL
hAif->linkConfig[0].outboundDataWidth = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[0].inboundDataType  = DATA_TYPE_DL
hAif->linkConfig[0].inboundDataWidth = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[0].multiRate         = 1
hAif->AxConfig[0].sampleRate          = AIF_SRATE_30P72MHZ
hAif->AxConfig[0].cpriPackMode        = AIF2_LTE_CPRI_8b8
hAif->AxConfig[1].sampleRate          = AIF_SRATE_30P72MHZ
hAif->AxConfig[1].cpriPackMode        = AIF2_LTE_CPRI_8b8
hAif->AxConfig[2].sampleRate          = AIF_SRATE_15P36MHZ
hAif->AxConfig[2].cpriPackMode        = AIF2_LTE_CPRI_4b4
hAif->AxConfig[3].sampleRate          = AIF_SRATE_15P36MHZ
hAif->AxConfig[3].cpriPackMode        = AIF2_LTE_CPRI_4b4
```

LTE5 limitation and workaround

During the verification of dual LTE configurations on a single CPRI link, it was found that AIF2 has limitations when LTE5 is combined with other LTE sampling rates (LTE10, LTE20).

Problem: there is a counter per PD channel which tracks the samples/32bitWords. The counter starts at zero on each CPRI basic frame. There is 1 CPRI basic frame per 260ns of time and the number of samples per basic frame depends on link rate. The problem is that there is only 1 shared counter per PD_link. So it is possible to share a PD_link between LTE10/LTE20 antenna containers because both have nx4

samples per basic frame. But no other radio standards can share the same PD_link. The result is that one of the radio standards will have clipped samples.

Workaround: It uses the link forwarding feature of AIF2 and it also relies on the fact that there are spare CPRI links in the system (AIF2 has a total of 6 links). The workaround consists in forwarding linka to linkb, using linkb in SerDes loopback mode, and then processing the protocol decoding of the first LTE sampling rate on linka while processing the protocol decoding of the LTE5 sampling rate on linkb.

The link forwarding delay is ~260ns.

Here is an illustration of the workaround using LTE10+LTE5:

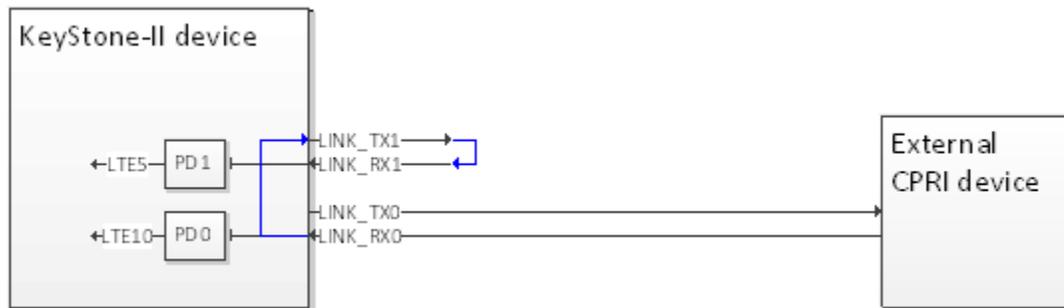


Figure 14: dual LTE workaround for LTE 5 MHz

From an AIF2 LLD standpoint, to enable this workaround (whenever LTE5 is mixed with LTE20 or LTE10), the user application needs to add the below extra AIF2 LLD configuration for link forwarding on one of the spare AIF2 links.

For instance, consider incoming CPRI traffic on link0 (dual LTE) and link1 from external devices, the user application could then configure link forwarding between link0 and link2 (spare link).

Consider a dual carrier system that needs to configure AIF2 for 2xLTE20+2xLTE5 on CPRI 8x link index 0. Then the LLD instance requires the following parameter settings to enable the workaround on spare link index 2:

```
hAif->linkConfig[0].numPeAxC          = 4
hAif->linkConfig[0].numPdAxC          = 2
hAif->linkConfig[0].linkEnable        = 1
hAif->linkConfig[0].linkRate          = AIF2FL_LINK_RATE_8x
hAif->linkConfig[0].outboundDataType  = DATA_TYPE_DL
hAif->linkConfig[0].outboundDataWidth = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[0].inboundDataType   = DATA_TYPE_DL
hAif->linkConfig[0].inboundDataWidth  = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[0].multiRate         = 1
hAif->linkConfig[2].numPeAxC          = 0
hAif->linkConfig[2].numPdAxC          = 2
```

```

hAif->linkConfig[2].firstPdAxC          = 2
hAif->linkConfig[2].linkEnable          = 1
hAif->linkConfig[2].linkRate             = AIF2FL_LINK_RATE_8x
hAif->linkConfig[2].nodeTx               = 1
hAif->linkConfig[2].nodeRx               = 0
hAif->linkConfig[2].outboundDataType     = DATA_TYPE_DL
hAif->linkConfig[2].outboundDataWidth    = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[2].inboundDataType      = DATA_TYPE_DL
hAif->linkConfig[2].inboundDataWidth     = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[2].multiRate            = 1
hAif->linkConfig[2].comType               = AIF2_LOOPBACK
hAif->linkConfig[2].RtEnabled             = 1
hAif->linkConfig[2].RtLinkRout           = (Aif2FI_LinkIndex) 0
hAif->AxConfig[0].sampleRate              = AIF_SRATE_30P72MHZ
hAif->AxConfig[0].cpriPackMode            = AIF2_LTE_CPRI_8b8
hAif->AxConfig[1].sampleRate              = AIF_SRATE_30P72MHZ
hAif->AxConfig[1].cpriPackMode            = AIF2_LTE_CPRI_8b8
hAif->AxConfig[2].sampleRate              = AIF_SRATE_7P68MHZ
hAif->AxConfig[2].cpriPackMode            = AIF2_LTE_CPRI_2b2
hAif->AxConfig[3].sampleRate              = AIF_SRATE_7P68MHZ
hAif->AxConfig[3].cpriPackMode            = AIF2_LTE_CPRI_2b2

```

Using the LLD instance configuration, link2 is used in internal loopback mode to handle the ingress LTE5 CPRI traffic coming from link0.

7.2 Dual mode

It is also possible with AIF2 to mix LTE and WCDMA traffic on a single CPRI link. The LLD supports this mode and needs to be carefully configured.

Some AIF2 facts

- Data format: a CPRI link must be all 7/15bit –OR- 8/16bit, so the formats cannot mix.
- CPRI Protocol Decoder/Encoder Link modules are not capable of mixing WCDMA UL and WCDMA DL on same link and direction. This means for a given link/PD or link/PE, all traffic MUST be 7bit, 8bit, 15bit, OR 16bit.

LLD implementation

Let's assume WCDMA DL is on 15-bit data width and UL on 7-bit data width. LTE DL and UL format is 15-bit. Since LTE and WCDMA UL formats are different, 2 separate links are used for the ingress traffic. The 2 links (assuming 0 and 1 here) are configured as:

Link0: Wcdma DL + LTE traffic, this link is connecting to the RF module

Link1: Wcdma UL traffic, the AIF2 RT module retransmits Link0 ingress traffic to Link1, Link1 is set in SerDes loopback mode.

Having both LTE DL and WCDMA DL on the same link protocol encoder is allowed:

- Both traffic use the same 15-bit format
- The 2 traffic can be separated in 2 different framing groups
- An appropriate dbmx pattern can map the 2 types of channel on given Cpri slot positions

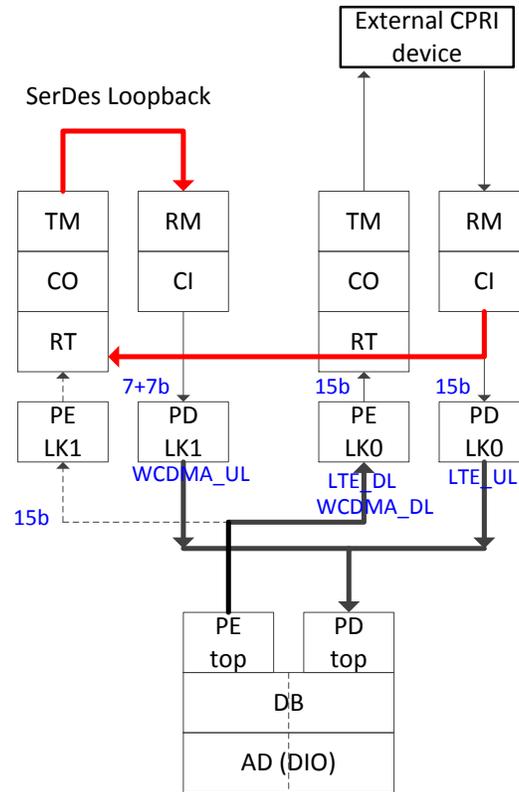


Figure 15: LLD implementation of dual mode on a single CPRI link

Container and bit mapping in the CPRI frame

Here is how the LLD formats the content of the CPRI basic frames for such a configuration.

- CPRI 8x rate
- 2xLTE20 antenna containers with AIF2_LTE_CPRI_8b8 packing mode
- 2xWCDMA antenna containers with 1 IQ sample per CPRI frame

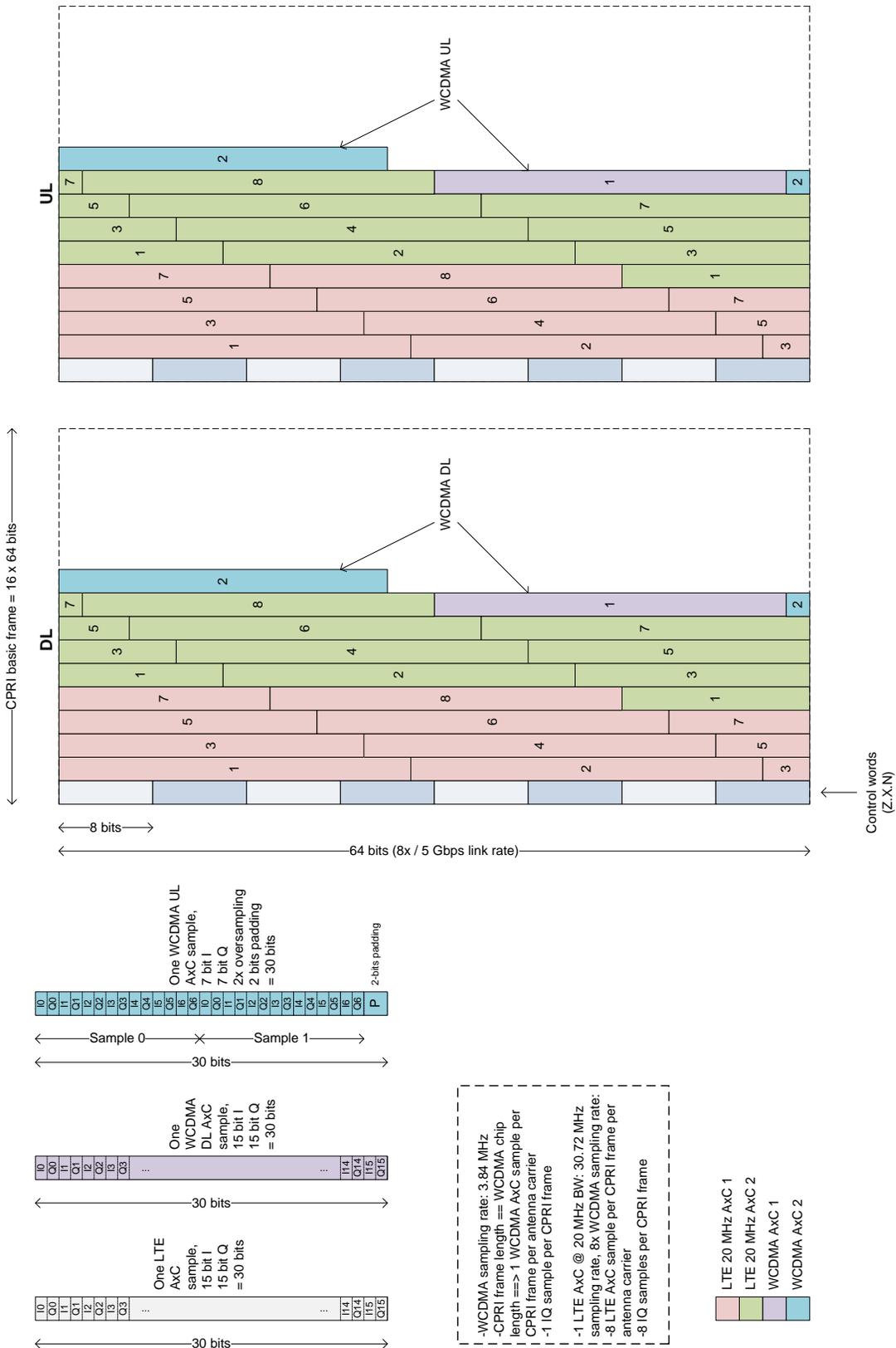


Figure 16: CPRI basic frame format for dual mode on a single CPRI link

Dual LTE/WCDMA configuration example

Consider a dual carrier system that needs to configure AIF2 for 2xLTE20+2xWCDMA on CPRI 8x link index 0. Then the LLD instance requires the following parameter settings to enable the implementation described earlier where a spare link index 2 (for instance) is used:

```
hAif->linkConfig[0].numPeAxC          = 4
hAif->linkConfig[0].numPdAxC          = 2
hAif->linkConfig[0].linkEnable        = 1
hAif->linkConfig[0].linkRate          = AIF2FL_LINK_RATE_8x
hAif->linkConfig[0].outboundDataType  = DATA_TYPE_DL
hAif->linkConfig[0].outboundDataWidth = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[0].inboundDataType   = DATA_TYPE_DL
hAif->linkConfig[0].inboundDataWidth  = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[0].dioEngine         = AIF2FL_DIO_ENGINE_0
hAif->linkConfig[0].multiRate         = 1
hAif->linkConfig[2].numPeAxC          = 0
hAif->linkConfig[2].numPdAxC          = 2
hAif->linkConfig[2].firstPdAxC        = 2
hAif->linkConfig[2].linkEnable        = 1
hAif->linkConfig[2].linkRate          = AIF2FL_LINK_RATE_8x
hAif->linkConfig[2].nodeTx            = 1
hAif->linkConfig[2].nodeRx            = 0
hAif->linkConfig[2].outboundDataType  = DATA_TYPE_DL
hAif->linkConfig[2].outboundDataWidth = AIF2FL_DATA_WIDTH_15_BIT
hAif->linkConfig[2].inboundDataType   = DATA_TYPE_UL
hAif->linkConfig[2].inboundDataWidth  = AIF2FL_DATA_WIDTH_7_BIT
hAif->linkConfig[2].dioEngine         = AIF2FL_DIO_ENGINE_1
hAif->linkConfig[2].multiRate         = 1
hAif->linkConfig[2].comType           = AIF2_LOOPBACK
hAif->linkConfig[2].RtEnabled         = 1
hAif->linkConfig[2].RtLinkRout       = (Aif2Fl_LinkIndex) 0
hAif->AxConfig[0].sampleRate          = AIF_SRATE_30P72MHZ
hAif->AxConfig[0].cpriPackMode        = AIF2_LTE_CPRI_8b8
hAif->AxConfig[1].sampleRate          = AIF_SRATE_30P72MHZ
hAif->AxConfig[1].cpriPackMode        = AIF2_LTE_CPRI_8b8
hAif->AxConfig[2].sampleRate          = AIF_SRATE_3P84MHZ
hAif->AxConfig[3].sampleRate          = AIF_SRATE_3P84MHZ
hAif->dioConfig[0].mode                = LTE;
hAif->dioConfig[0].numPeDBCH          = 2;
hAif->dioConfig[0].numPdDBCH          = 2;
hAif->dioConfig[0].offsetPeDBCH       = 0
hAif->dioConfig[0].offsetPdDBCH       = 0;
hAif->dioConfig[0].numLink            = 1;
```

```

hAif->dioConfig[0].firstLink      = 0;
hAif->dioConfig[1].mode            = WCDMA;
hAif->dioConfig[1].numPeDBCH      = 2;
hAif->dioConfig[1].numPdDBCH      = 2;
hAif->dioConfig[1].offsetPeDBCH   = 2;
hAif->dioConfig[1].offsetPdDBCH   = 2;
hAif->dioConfig[1].numLink        = 1;
hAif->dioConfig[1].firstLink      = 0;

```

8. Software implementation of CPRI FastCM over AIF2

8.1 Introduction

AIF2 supports 4B/5B Fast Ethernet, which is commonly used to configure and control RF remote radio heads (RRH). 4B/5B encoding is happening on top of the SerDes 8B/10B encoding. AIF2 HW supports 8B/10B encoding at the PHY layer and 4B/5B encoding at the Protocol layer.

Ethernet frame structure:

| Preamble | Start of Frame | Dst Adr | Src Adr | Length | Payload | CRC |
|----------|----------------|---------|---------|---------|-------------|--------|
| 7 bytes | 1 bytes | 6 bytes | 6 bytes | 2 bytes | 1-1500 byte | 4 byte |

Figure 17: Ethernet frame structure

CPRI start of Frame characters (JK) and End of Frame characters (TR) is added to the 4B5B encoded data, and transmitted by AIF2.

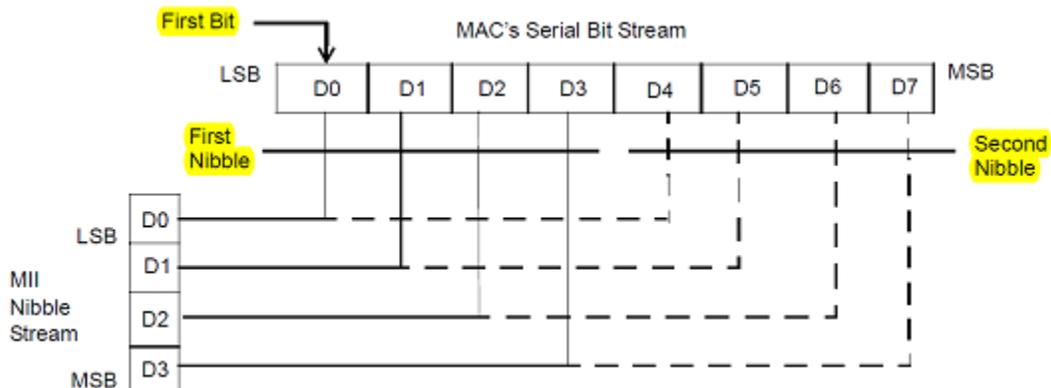


Figure 18: IEEE 802.3 data transmission format

- 1) But AIF2 HW follows the big endian convention and transmits the MSB of the second nibble, this is the violation of the spec and documented in errata for Keystone-I and Keystone-II devices.
- 2) JK start of preamble sequence for CPRI has to be inserted at the First octet by AIF2 to form essentially 8 Octets of preamble , but AIF2 hardware appends this preamble making 9 Octets of preamble. This is also violation of the IEEE 802.3 spec.

Two solutions to overcome this limitability of the AIF2 hardware are proposed:

Sol 1: to make the changes in RRH HW (FPGA for instance) so that it expects data to be received in the reverse sequence and scans for a 9-Octet preamble, instead of a 8-Octet one.

Sol 2: to implement the 4B5B encoder and decoder in DSP or ARM cores, i.e. a software implementation of Fast CM and transmits data over AIF2 CPRI control stream HW in Null Delimiter mode.

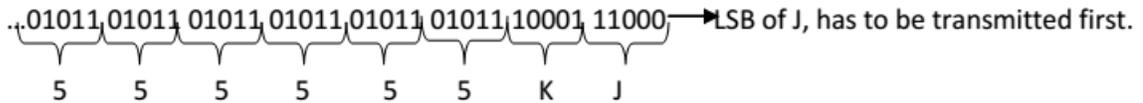
8.2 Software workaround implementation

This documents talks about the software implementation of the 4B5B encoder and decoder. The implementation can be found in .\ti\drv\aiif2\test\cprifastcm for reference. Consider the following preamble of the following stream at MAC level:

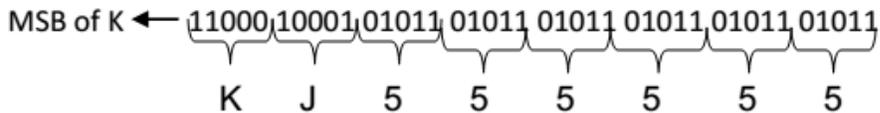
| EtherFrame | | 4B (LSB First) <i>MI I Layer</i> |
|------------|-----|---------------------------------------|
| Preamble | 55h | 5 |
| | | 5 |
| | 55h | 5 |
| | | 5 |
| | 55h | 5 |
| | | 5 |
| | 55h | 5 |
| | | 5 |
| | 55h | 5 |
| | | 5 |
| SFD | D5h | 5 |
| | | D |

Figure 19: Ethernet frame preamble

Data that is been expected to be transmitted over CPRI is as follows:



The AIF2 hardware in 4B5B mode transfers the data in the following format:



So to correct this misbehavior, DSP software is used to for the 4B to 5B encoding on the Egress side, and 5B to 4B decoding on the Ingress side.

8.2.1 AIF2 configuration in Null delimiter mode

AIF2 is configured to work in Null Delimiter mode, with 0xFF as the Null Delimiter. AIF2 in this mode just As per the CPRI spec the 5B character “11111” is used as idle character ,so on the transmit side once the SSD symbols are inserted by the 4B5B encoder into the CPRI stream AIF2 hardware inserts the idle character and transmits the data. The AIF2 PdLink setup and PeLinkSetup configurations for NULL delimiter mode is as follows

```

PdLinkSetup.bEnablePdLink = TRUE;
PdLinkSetup.CpriEnetStrip = 0x0; //Disable Ethernet strip for control channel
PdLinkSetup.Crc8Poly = CRC8_POLY;
PdLinkSetup.Crc8Seed = CRC8_SEED;
PdLinkSetup.CpriCwNullDelimiter = 0xFF;
PdLinkSetup.CpriCwPktDelimiter[0] = AIF2FL_CW_DELIM_NULLDELM;
PdLinkSetup.PdCpriCrcType[0] = AIF2FL_CRC_32BIT;
PdLinkSetup.bEnableCpriCrc[0] = FALSE; //Disable CPRI CRC for control channel 0
PdLinkSetup.PdPackDmaCh[0] = 124; //Set DB channel 124 as a dma ch for control channel 0
PdLinkSetup.bEnablePack[0] = TRUE; //disable CPRI control channel 0 packing
PeLinkSetup.CpriCwNullDelimiter = 0xFF; //K 27.7 character
PeLinkSetup.CpriCwPktDelimiter[0] = AIF2FL_CW_DELIM_NULLDELM;
PeLinkSetup.PePackDmaCh[0] = 124;
PeLinkSetup.bEnablePack[0] = TRUE;

```

8.2.2 TX 4B 5B encoder

On the egress side before the data is being pushed to the Transmit queue, a 4B to 5B encoder function is called. The 4B5B encoder kernel expects the data, in the following format:

| Preamble | Start of Frame | Dst Adr | Src Adr | Length | Payload | CRC |
|----------|----------------|---------|---------|---------|-------------|--------|
| 7 bytes | 1 bytes | 6 bytes | 6 bytes | 2 bytes | 1-1500 byte | 4 byte |

Generally used preamble for the Ethernet frame: 0x55, 0x55, 0x55, 0x55, 0x55, 0x55, 0x55, 0x55 and the Start of Frame is 0xD5. The 4B5B encoder does the following data manipulation to the input stream to form a packet as per the Fast CM specification:

- Each 4Bit Nibble is converted into a 5Bit Nibble.
- Insert SSD i.e JK into the first symbol location of the Preamble
- Append the ESD(TR) at the END of the stream after the CRC .

4B5B output frame structure:

| Preamble | Start of Frame | Dst Mac Addr | Src Mac Addr | Length | Payload | CRC | ESD |
|---|----------------|--------------|--------------|---------|---------------------|--------------|-----------------|
| 10 Bits SSD(JK) 60 Bits Preamble | 1*10 Bits | 6*10 Bits | 6*10 Bits | 20 Bits | (1-1500)*10 Bits | 4*10 Bits | 10 Bits (TR) |

The 4B5B encoder function kernel is implemented using the lookup table approach, each 4Bit Nibble is used as an Index in the Lookup table and corresponding 5Bit encoded value is loaded using the following table from AIF2 user's guide:

Table 5-2 Fast Ethernet 4B/5B Encoding (Part 1 of 2)

| Name | 4b | 5b | Description |
|------|------|-------|-------------|
| 0 | 0000 | 11110 | hex data 0 |
| 1 | 0001 | 01001 | hex data 1 |
| 2 | 0010 | 10100 | hex data 2 |
| 3 | 0011 | 10101 | hex data 3 |
| 4 | 0100 | 01010 | hex data 4 |
| 5 | 0101 | 01011 | hex data 5 |
| 6 | 0110 | 01110 | hex data 6 |
| 7 | 0111 | 01111 | hex data 7 |
| 8 | 1000 | 10010 | hex data 8 |
| 9 | 1001 | 10011 | hex data 9 |
| A | 1010 | 10110 | hex data A |
| B | 1011 | 10111 | hex data B |

Table 5-2 Fast Ethernet 4B/5B Encoding (Part 2 of 2)

| Name | 4b | 5b | Description |
|------|--------|-------|-------------|
| C | 1100 | 11010 | hex data C |
| D | 1101 | 11011 | hex data D |
| E | 1110 | 11100 | hex data E |
| F | 1111 | 11101 | hex data F |
| I | -NONE- | 11111 | Idle |
| J | -NONE- | 11000 | SSD part1 |
| K | -NONE- | 10001 | SSD part2 |
| T | -NONE- | 01101 | ESD part1 |
| R | -NONE- | 00111 | ESD part2 |
| H | -NONE- | 00100 | Halt |

8.2.3 RX 5B 4B Decoder

On the Ingress side, when the packets are received software needs to scan for the Start of Frame SSD and the END of frame ESD and then send the received packet to the decoder. On the Rx side, if there is 0xFF byte in the data that has been transmitted by the RRH, then the AIF2 fragments into multiple packets at each 0xFF boundary. The application software provided part of the example with this application report , defragments the multiple packets into a single packet and sends it to the decoder for further processing.

Consider the hypothetical case, of data being transmitted from the RRH to the Baseband board:

| | | | | | | | | | | | | | | | | | | | |
|-----|-----|-----|-----|-----|----|----|----|------|------|------|----|----|----|----|------|------|------|----|-----|
| SSD | 0x1 | 0x2 | 0x3 | 0x4 | .. | .. | .. | 0xFF | 0x10 | 0x11 | .. | .. | .. | .. | 0xFF | 0x20 | 0x21 | .. | ESD |
|-----|-----|-----|-----|-----|----|----|----|------|------|------|----|----|----|----|------|------|------|----|-----|

When such a packet is received at the AIF2, AIF2 segments it into 3 fragments as shown below:

| | | | | | | | |
|------|------|-----|-----|-----|----|----|----|
| SSD | 0x1 | 0x2 | 0x3 | 0x4 | .. | .. | .. |
| 0x10 | 0x11 | .. | .. | .. | .. | .. | |
| 0x20 | 0x21 | .. | ESD | | | | |

The application code in the software provided, scans for the SSD and stitches all the packets into a single packet till it finds ESD. Once the SSD and ESD are detected and validated the packet is further sent to a 5B to 4B decoder which decodes the RX packet and forms a Ethernet Packet for further processing at the MAC Layer.